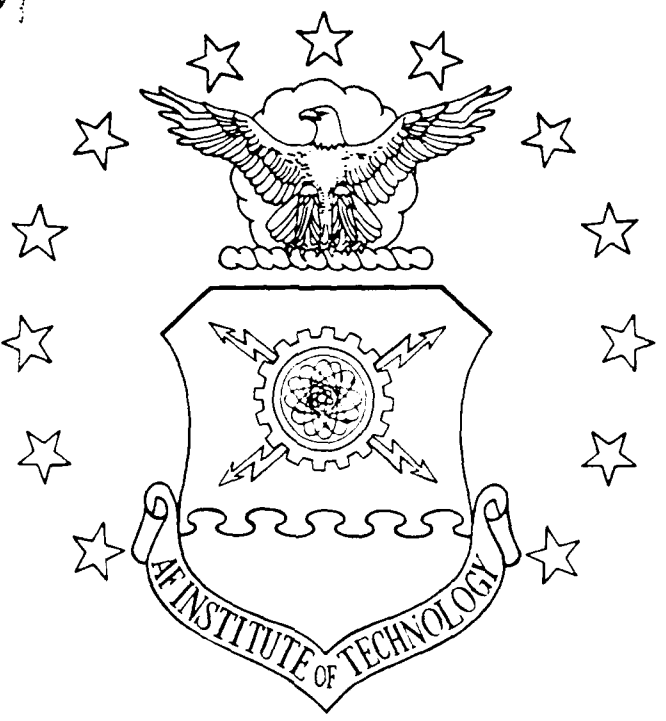


AN OBJECT ORIENTED ANALYSIS METHOD FOR Ada AND EMBEDDED SYSTEMS

THESIS

Steven G. March
Captain, USAF

AFIT/GCS/ENC/89D-1

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCS/ENC/89D-1

AN OBJECT ORIENTED ANALYSIS METHOD FOR Ada AND EMBEDDED SYSTEMS

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air University
In Partial Fulfillment of the
Requirements for the Degree of
Master of Science in Computer Engineering

Steven G. March, B.S.
Captain, USAF

December, 1989

Approved for public release; distribution unlimited

Acknowledgments

This thesis could not have been possible without the help and encouragement of a number of people. First, I'd like to thank my thesis advisor, Maj David Umphress, for his guidance, patience, and understanding during the course of this research. Our discussions gave me the insight and perspective to pursue what at times seemed an impossible goal. I would also like to thank my committee members, Lt Col John Valusek and Maj James Howatt for their constructive critiques and ideas.

A special word of thanks goes out to Don Principiotta. His friendship and interest was an invaluable asset in the form of research leads and honest criticism.

Finally, but most importantly, my deepest gratitude to my wife, Dianne. Her constant encouragement, support, and understanding enabled me to apply the effort needed to accomplish this thesis.

Steven G. March

Table of Contents

List of Figures

List of Tables

Abstract

Object-Oriented Design (OOD) has become a popular approach to software development with Ada. One of the difficulties in applying OOD is that the information available to the designer (the product of requirements analysis) is typically presented in a form inappropriate to OOD. Traditional requirements analysis tools (e.g. data flow diagrams) organize the software requirements based upon the *functions* the system must perform. Recent research suggests that an *object-oriented* approach to requirements analysis is a more natural lead-in to OOD.

The goal of this thesis was to define the tools, steps, and heuristics for an object-oriented analysis (OOA) method of modeling software requirements. The choice of tools used to capture the requirements makes the method particularly suitable for use when developing embedded systems. The method emphasizes communication with both the domain expert and the designer.

The OOA method consists of two phases. The objective of the first phase is to capture the software requirements using unstructured tools such as concept maps, storyboards, and a list of external events to which the system must respond. The second phase involves structuring these requirements into a model based upon the software objects.

The thesis also addressed the possibility of automated support for the OOA method, and proposes an OOA tool to assist the analyst. The OOA method was applied to a sample requirements analysis problem to demonstrate the method's feasibility.

AN OBJECT ORIENTED ANALYSIS METHOD FOR Ada AND EMBEDDED SYSTEMS

I. Introduction

The object-oriented approach to software development is entering its adolescence. Discussions about the object-oriented paradigm now routinely appear in the computing literature, as practitioners recognize its potential benefits in developing reliable, maintainable software. This thesis examines the application of object-oriented techniques during the requirements analysis phase of software development.

1.1 Background

In the 1970s, the Department of Defense (DoD) experienced first-hand the symptoms of the *software crisis*: unpredictable development costs and schedules, poor software reliability, costly maintenance, and delivered software failing to meet the needs of the user [Booch, 1983:6-7]. As one step toward resolving this predicament, the DoD sponsored the design of a new programming language, Ada, to be used for the development of embedded systems applications for the military. The Ada language includes features (e.g. strong typing, packages, and tasks) which aid software engineers in managing the complexity of large software systems.

The availability of these modern language constructs in Ada required software engineers to formulate new methods to utilize them. Russell Abbott [Abbott, 1983] and Grady Booch [Booch, 1983] introduced the use of object-oriented techniques as a means of applying the new features of Ada to combat software complexity. The structure of software developed with an object-oriented approach is patterned on the *objects* evident in the real-world problem. A designer creates software entities to

implement these objects. The resulting similarity between the structure of the problem and that of the solution tends to produce a more natural design than a design mapped only into the predefined data and control structures of a typical programming language. The object-oriented approach to software development is unique in its ability to support the principles of abstraction, information hiding, and modularity [Pressman, 1987:334] which lead to more understandable and maintainable software.

This original Object-Oriented Design (OOD) method called for the development of a textual *informal strategy* of the solution to be used as the foundation for the design phase. From this informal strategy, the designer identified the relevant objects and operations required for the solution of the problem. Unfortunately, neither Abbott nor Booch gave much guidance for writing the informal strategy. In practice, the typical result is a vague, unstructured paragraph which serves as a frail basis for the design.

Recent research suggests the use of object-oriented techniques in the earlier phase of requirements analysis provides a more coherent approach to object-oriented development [Pressman, 1987, EVB, 1989, Ladden, 1989]. A complete life cycle object-oriented methodology provides a stronger framework for the application of Ada in the management of software complexity.

A recent thesis by Capt Patrick Barnes [Barnes, 1988] proposed an Object Oriented Design methodology based on the concepts of decision support systems. Barnes suggests that a graphical concept map may be a better means of describing the solution strategy than Booch's informal strategy method [Barnes, 1988:3-5], and recommends research into the use of concept maps in the requirements phase. This thesis further explores the idea of using graphical object-oriented tools in the requirements analysis phase as a substitute for the textual informal strategy.

1.2 Problem Definition

The objective of this thesis is to develop an Object Oriented Analysis (OOA) method to model software requirements. This analysis method will provide guidelines for identifying the objects in the problem space, their attributes, and the relationships among the objects. The specific objectives of this thesis are to

a. Determine the requirements of an OOA method.
b. Define the steps of an OOA method to represent the problem space.
c. Identify the requirements for a software tool to assist an analyst in applying the OOA method.
d. Validate the OOA method by applying it to a sample requirements analysis problem.

1.3 Scope

The requirements analysis method developed in this thesis assumes that the analyst has expended prior effort in defining the overall *system* requirements. Therefore, the method considers only the software component of a larger system. The method also ignores the analysis of factors such as cost, effort, schedule, or testing; only the behavioral requirements of the software are addressed.

The method concentrates mainly on the analysis, rather than the determination, of requirements. In other words, the method assumes that the user has made an effort to *identify* the needs of the system, and concentrates on *capturing* and *documenting* these requirements in a model which forms the basis for the software design phase. In reality, it is often difficult to separate requirements determination and analysis into distinct phases. Therefore, the techniques of any analysis method may have to be applied iteratively until a satisfactory set of requirements can be defined. However, tools such as rapid prototyping, which are helpful for requirements determination, are not specifically included in the OOA method. The result of the

OOA method is a model of the system which can be understood by both the user and the designer.

Since Ada was originally developed to program embedded systems, the OOA method will concentrate on this arena as well. These systems seem well suited to object-oriented design, with concrete physical objects that require modelling in software. Embedded systems also tend to have requirements that can be identified early in the life cycle, with a fairly well defined hardware/software interface. Although the OOA method is aimed primarily towards embedded systems, many of the concepts presented here apply as well to general software development. However, the method provides only limited support for identifying reusable objects or potentially concurrent objects and operations.

During the development of the OOA method, consideration was given to the complexities associated with specifying *large* software systems. As the complexity and size of the software increases, it becomes important to view the software from various levels of abstraction. The OOA method therefore should support the layering of objects into hierarchical levels of abstraction.

The method assumes that object-oriented techniques will be used in the design phase, and that Ada will be the implementation language. Ada is not a "true" object-oriented language in that it doesn't fully support concepts such as inheritance and dynamic binding; however, Ada can still support a form of object oriented programming. The OOA method addresses object oriented concepts as Ada supports them. Given these restrictions, the analyst can use the OOA method in combination with OOD when the implementation language is one that supports data encapsulation and information hiding, such as Ada or Modula-2.

1.1 Approach and Overview

This research project consisted of four phases:

1. An investigation of current analysis techniques.
2. Formation of the tools and steps of the OOA method.
3. Description of an OOA tool to assist the analyst.
4. Validation of the method.

These phases were somewhat overlapping. Examining current analysis techniques prompted new ideas about the requirements and design of the OOA method. Also, the process of developing the OOA method naturally identified many of the requirements for the corresponding tool.

1.4.1 Review of Current Analysis Techniques. The first step in formulating the OOA method was a review of the literature to identify the current state of practice. Since at the outset of this project little research had been accomplished in the application of object-oriented techniques to software requirements analysis, the review covered a number of topics surrounding the issue. The first subject was the application of object-oriented techniques during the later phase of software design. The second area addressed was the software definition phase. This examination identified the information captured during requirements analysis, and the major tools and approaches used to capture this information. Finally, the review considered the attempts that have been made in applying the object-oriented paradigm to the definition phase. Chapter II of this thesis discusses the results of this literature review.

1.4.2 OOA Method Development. The requirements for the object-oriented analysis method were based on the information gathered during the literature review. The review of the OOD process identified the information needed in the requirements model to successfully apply object-oriented design. The specific tools and steps of the OOA method were then developed from these requirements. The requirements for, and steps of, the OOA method are described in Chapter III.

A good deal of creativity was required to conceive the steps of the OOA method. A promising starting point was to first conceive the end components of the object-oriented model of the software requirements. These model components were selected to represent all of the information required for an object-oriented design. Another factor influencing the tool selection was the need to define a model which could be easily understood by both the users and developers of the software.

With this end result defined, attention was then directed toward identifying the steps and heuristics that go into developing the model components. These guidelines were defined in sufficient detail to enable an analyst to identify the objects and operations of the problem at various levels of abstraction.

1.4.3 Requirements for an OOA Tool. As the steps and tools of the OOA method were defined, consideration was given to their automated support. Particular attention was given to the representations, operations, memory aids, and control aids (ROMC–see [Sprague and Carlson, 1982]) of a tool to support the OOA method. Concept maps identified the elements of the OOA method and the relationship between the method steps. Once the potential support areas were identified, the proposed OOA tool was described through story boards. The description of an object-oriented analysis tool is outlined in Chapter IV.

1.4.4 Method Validation. The concepts defined in OOA method were validated through application of the method on a sample problem. The problem selected to validate the method needed to be of sufficient size and complexity to give a reasonable demonstration of the method, yet small enough to handle in an academic environment. The classic "elevator problem" was chosen for the task of evaluating the method. This problem, originally used in a workshop sponsored by the Association of Computing Machinery (ACM) [Yourdon, 1989:631], calls for the analysis of the software requirements for an elevator control system for a building with four elevators serving 10 floors.

The tools and guidelines of the OOA method were applied to analyze and model the software requirements for the elevator control system. The results of this exercise were compared against the method goals identified in section 3.1. Also, since the elevator problem has been previously analyzed using different requirements analysis methodologies, the application of the OOA method on this same problem enabled a comparison to be made between the OOA method and previous, function based analysis methods. The results of this evaluation are described in Chapter V.

1.5 Maximum Expected Gain

The result of this research is a method defining the steps needed to model the problem space in an object oriented manner. This model provides a more straightforward lead-in to Object-Oriented Design than current function-based tools such as data flow diagrams, thus enabling a better object-oriented design. The method also provides the analyst with more guidelines and structure than the informal strategy of Booch [Booch, 1983] and Abbott [Abbott, 1983], while still retaining a more unstructured communication with the domain experts.

A more straightforward method of applying object oriented techniques in the software analysis phase should result in wider research and application of the object-oriented paradigm. Additionally, the OOA method and tool could be used to support instruction in requirements analysis, OOD, and the proper use of Ada constructs.

1.6 Sequence of Presentation

The chapters of this thesis follow the phases of research identified in the *Approach and Overview* section. Chapter II lays the foundation with a review of the current literature in the broad areas of requirements analysis and the object-oriented paradigm. Chapter III identifies the requirements, tools, and steps of the Object-Oriented Analysis method. Chapter IV provides an outline for a tool to support the analyst in applying the OOA method. Chapter V provides an evaluation of the

method by applying it to a sample requirements analysis problem. Finally, Chapter VI identifies conclusions gathered from this research and recommendations for further study.

II. Literature Survey

The growing number of articles, books, and object-oriented languages in recent years implies that the object-oriented paradigm is more popular than ever. The trend in the literature also suggests an expansion of the paradigm from the coding and designing activities into the earlier activity of requirements analysis. This chapter begins by discussing the application of object-oriented techniques during the design phase. It then reviews popular approaches to requirements analysis in order to identify the tools and methods used in capturing software requirements. The chapter closes with a discussion of recent attempts to apply object-oriented techniques to requirements analysis.

2.1 Object-Oriented Techniques in the Design Phase

One of the perceived benefits of the object-oriented paradigm is its application of modern software engineering principles to deal with the complexity of large problems. The result is a more natural mapping between the real world problem (the "problem space") and the solution represented by the software.

Other popular design methods tend to carve the architecture of software systems along either functional or data-structure lines [Booch, 1987b:37] These methods work fine when used with older languages whose primary structuring mechanism is a procedure. However, these methods fail to utilize the structuring capabilities of newer languages, such as the Ada package construct or the object structure of Smalltalk, that aid in the management of complexity [Booch, 1987b:37]. Pressman summarizes the promising aspects of object-oriented design (OOD):

> The unique nature of object-oriented design lies in its ability to build upon three important software design concepts: abstraction, information

> hiding, and modularity. All design methods strive for software that exhibits these fundamental characteristics, but only OOD provides a mechanism that enables the designer to achieve all three without complexity or compromise. [Pressman, 1987:334]

This section will define an object model to be used throughout the remainder of the thesis, and portray object-oriented design (OOD) as it applies to programming in Ada.

2.1.1 Object Model. An object-oriented perspective views the world in terms of objects and behaviors. Work is accomplished when an object sends a message to another object, asking it to perform some behavior. Each object maintains some state information which may be updated when an object performs a behavior [Barnes, 1988:2.13]. The state of higher level composite objects can be described in terms of the state of each of its component objects.

The definitions of terms such as object and behavior differ somewhat among authors. There seems to be three major reasons for the differences: the domain of use (information systems vs embedded systems), the purpose of the discussion (practical vs theoretical), and the support of the language used to implement the concepts.

In an embedded systems view of OOD, the objects come primarily from the physical entities of the problem. Most objects model the state of each of these physical entities. Other types of systems take a more liberal view of an object. Information systems often expand the view of an object to entail relationships between data items, identified when the data model is normalized. Shlaer and Mellor include in their object definition any abstract concept in the real world [Shlaer and Mellor, 1989:67]. They go so far as to identify different types of objects:

- Abstractions of *tangible things* from the real world.
- Abstractions of *roles* of entities.

- Abstractions of *specifications or quality criteria.*
- *Collections or aggregations* of tangible items.
- *Steps* in the execution of a process.

An object model defined by Bralick [Bralick, 1988] provides a more theoretical foundation for the paradigm. However, Barnes identifies two weaknesses in using this model in design. First, it lacks an explicit method of describing the interactions of objects. Second, the model's flexibility in representing entities makes it too ambiguous to provide a designer with a clear path for design [Barnes, 1988:2.20]. Barnes defined an object model aimed at the design phase, based on the work by Bralick and a more restrictive model based on Smalltalk objects.

Presented here is the object model developed by Barnes. Since the emphasis of this thesis is toward object-oriented concepts as applied toward the Ada language, it seems fitting to also include characteristics of objects introduced by Grady Booch [Booch, 1986], who applied the *object-oriented paradigm* to development with Ada. Together, these descriptions will provide a basis for identifying the information that must be captured during requirements analysis.

2.1.1.1 An Object Model for Design. Barnes's object model is characterized by objects, classes of objects, operations, attributes, and relations among object classes [Barnes, 1988:2.25]. Barnes summarizes his object model in the following terms:

- An *object* is a unique entity defined by attributes which serve to identify the object and relations which associate it with other objects, attributes, and operations. Required attributes are name, behavior domain, and class. Relations include sets of operations, components, actors, and servers.
- An *attribute* identifies an object or operation.
- A *relation* represents an association of an object or operation with other system objects, operations, or relations.

- An *operation* is the description of how an object performs some behavior. Required attributes are name and algorithm. Relations include sets of actors, servers, arguments, and modified objects.
- A *class* is a complete design of an object which may be used as a template from which other objects derive their characteristic structure and function. [Barnes, 1988:3.1]

In his model, Barnes distinguishes between the terms "behavior" and "operation". He defines a behavior as a more informal description of an object's functionality in response to a stimulus from another object, while an operation is a more formal set of algorithms defining how the behavior is performed [Barnes, 1988:2.24]. Barnes also identifies the relationships between objects by classifying objects as actors or servers, depending on the direction the message is passed [Barnes, 1988:2.25].

This object model serves as a foundation for the characteristics of an object. More specific aspects of the paradigm as implemented by Ada are considered by including the definitions used by Booch.

2.1.1.2 Booch's Characteristics of an Object. Booch defines an object in general terms as "an entity whose behavior is characterized by the actions that it suffers and that it requires of other objects"[Booch, 1986:211]. He goes on to provide a more detailed analysis of an object's characteristics.

State. The state of an object denotes the value of the object at a point in time. This value includes the state of any sub-objects contained in the object in question [Booch, 1986:215]. For example, the state of a windowing environment includes the state of each window running in the environment

Actions and Objects. An object may require operations from other objects, or it may serve the requests of other objects. These operations on an object serve to modify or examine the state of the object. Booch identifies three classes of operations. *Constructor* operations modify the state of an object. *Selector* operations

examine the current state of an object. *Iterator* operations visit all sub-components of a complex object in turn.

An object may be classified by the way it relates to other objects. *Actor* objects operate on others but are not operated on. *Server* objects are operated on by others, but never inflict operations on others. *Agent* objects perform some operation on behalf of other objects. An agent is both the receiver and initiator of operations [Booch, 1986:216].

Classes of Objects. A class is a set of unique objects that share the same characteristics. Each object in the class has the same set of operations as other objects in the class. A class is characterized by a set of values, and a set of operations applicable to objects of that class [Booch, 1987a:21]. A system may possess a hierarchy of classes, where a *metaclass* defines a set of classes; however, Ada's support for implementing this hierarchy is somewhat limited. [Booch, 1986:216].

Object Names. Names are used to identify objects. Each object has at least one name, and could be referred to by multiple names (aliases) [Booch, 1986:216].

Visibility of Objects. Objects should be restricted in their visibility, so that they may collaborate only with other objects that are logically required to implement it's design. Unrestricted visibility allows any object to operate on any other object. Limiting the visibility makes the system more understandable and modifiable [Booch, 1986:216].

Views of Objects. An object may be viewed from two different perspectives: inside or outside the object. The outside view represents the abstract behavior of an object – its interface with the rest of the world. The inside view reveals the details of how the object and its operations are implemented [Booch, 1986:217]. Other objects only see the outside view of an object.

2.1.1.3 Summary. Together, Barnes' object model for design and Booch's characteristics of an object define a framework for an Ada view of the object-oriented paradigm. This model also serves as a starting point for identifying the information that must be captured during software requirements analysis. The next section discusses the object-oriented design method proposed for Ada by Grady Booch.

2.1.2 Object-Oriented Design (OOD). The foundations of applying the object-oriented paradigm to the design phase date back at least as far as Parnas' discussion of information hiding [Parnas, 1972]. However, it wasn't until the development of Ada and object oriented languages that interest mounted in applying the paradigm in the design phase. All languages are object-oriented to some degree [EVB, 1985, Bralick, 1988:2.2]. However, it is a language's support for data abstraction, information hiding, and to a lesser extent dynamic binding and inheritance that makes it suitable for object-oriented programming [Pascoe, 1986:140]. Although Ada is weak in the areas of dynamic binding and inheritance [Pascoe, 1986:142], it strongly supports the concepts of abstraction and information hiding, making it suitable for implementing an object-oriented design [Booch, 1986:216].

2.1.2.1 Booch's Object-Oriented Design Method. The version of object-oriented design (OOD) made popular by Grady Booch [Booch, 1983] relies on the research of Russell Abbott. Abbott's approach to software design began with the development of a textual "informal strategy" [Abbott, 1983]. Booch applied this approach in an object-oriented framework to develop his initial version of OOD.

"Traditional" OOD. Booch's initial version of OOD involved the following steps [Booch, 1983, EVB, 1985, Pressman, 1987]:

1) *Define the problem.* The use of analysis tools is appropriate at this point to define the problem space [Booch, 1983:11]. Some advocates

recommend stating the underlying problem in a single, grammatically correct sentence [EVB, 1985, Pressman, 1987].

2) *Develop an Informal Strategy.* English prose is used to define a solution using the terms of the problem space [Booch, 1983:42]. This informal strategy specifies the relationships among objects that make up the solution [EVB, 1985:1.4].

3) *Formalize the Strategy.* This step involves four sub-steps. First objects and their attributes are identified from the nouns and noun phrases in the informal strategy. Next, operations on the objects are identified from the verbs and verb phrases in the informal strategy. Third, interfaces between objects are established, and expressed in a graphical notation (see the example in figure 2.1). Finally, the operations are implemented, potentially applying the OOD process recursively on the operations. [Booch, 1983:42-43]

Booch and his followers have since modified their approach somewhat, resulting in less of a dependence on an informal strategy to identify objects and operations.

"Contemporary" OOD. The steps in this "contemporary" approach to OOD are listed below [Booch, 1986, Booch, 1987a, Booch, 1987b]:

1) *Identify the objects and their attributes.* Although nouns used to describe the problem space are mentioned as a possible guideline to identifying objects, no mention is made of developing an informal strategy[Booch, 1987b:48].

2) *Identify the operations.* This step characterizes the behavior of each object or class by identifying the operations that affect each object or class and the operations that each must initiate [Booch, 1987b:48-49].

3) *Establish the visibility of each object.* The static visibility of each object is defined in relation to other objects in the system, using the graphical notation identified earlier [Booch, 1987b:49].

4) *Establish the interface of each object.* The interface of each object or class is specified using some suitable notation, such as an Ada package specification. This interface is the view of the object or class to other objects in the software system [Booch, 1987b:49].

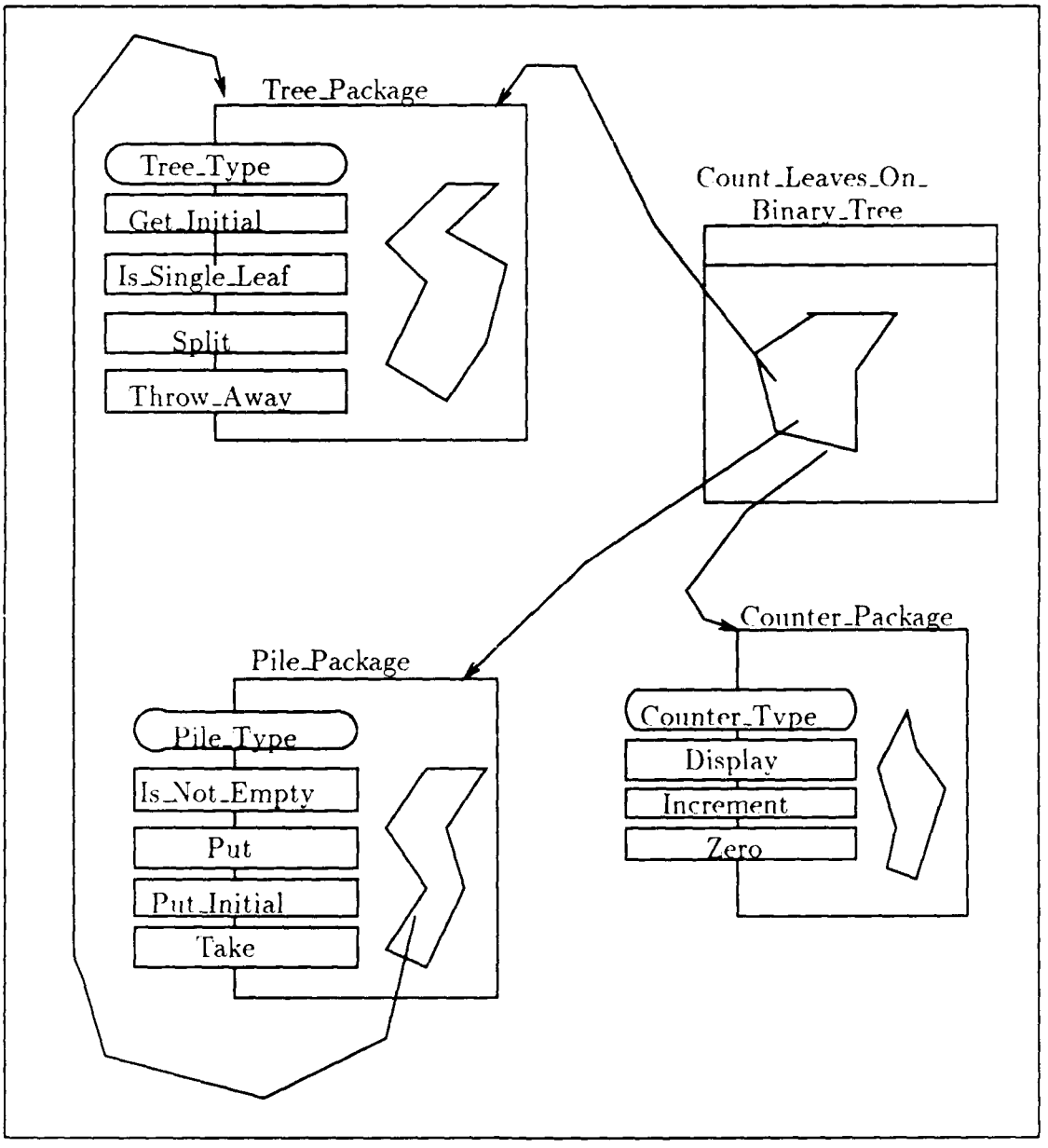


Figure 2.1. OOD Graphical Interface Diagrams ("Booch Blobs") [Booch, 1983:75]

5) *Implement each object.* Each object is implemented using a suitable language feature. The object may be further decomposed by recursively reapplying the OOD process, or composed in a bottom-up fashion from existing lower-level objects or classes [Booch, 1987b:49].

This version of OOD minimizes the emphasis on the informal strategy and, instead, emphasizes the use of traditional analysis tools to define the problem [Booch, 1986, Booch, 1987b, Booch, 1987a:47]. Ladden summarizes reasons for the shift away from the informal strategy approach to OOD:

> The viability of the well-known technique of developing an 'informal strategy' by creating a narrative description of the problem, and then selecting the objects, operations and attributes of the system from the nouns, verbs, adjectives, and adverbs of this narrative description is questioned. It inherently lacks rigor due to the impreciseness of the English language; the approach appears to have been disregarded by its originator; and its suitability for large projects has been criticized. [Ladden, 1989:87]

Object-Oriented Design as known today is not a full life cycle method [Booch, 1986:84]. OOD assumes that the problem space has been previously defined and organized using some form of analysis tools. Two approaches to this analysis are possible. A traditional analysis method may be applied to the problem and then translated into an object-oriented representation. An alternative is to use object-oriented techniques throughout the life cycle [EVB, 1989:15]. Both approaches will be considered in the last section of this chapter.

As defined in chapter I, the primary goal of this thesis is to develop an object-oriented approach to the analysis phase. This object-oriented requirements analysis (OORA) activity will precede the OOD process to achieve a more complete life cycle methodology. The development of this Object-Oriented Requirements Analysis method assumes than an object-oriented approach similar to the "contemporary" OOD approach will be used during the design phase.

2.2 *The Definition Phase*

As mentioned at the outset of this chapter, the object-oriented paradigm seems to be working its way backward in the life cycle. Whereas object-oriented techniques have been practiced in the development phase for nearly a decade (an eon in the rapidly advancing computer field!), application of the paradigm in the requirements analysis phase is a much more recent phenomenon. This section identifies current approaches to delineating requirements for software systems, presenting a menu of alternative representations on which an object-oriented approach may be based.

2.2.1 Software Requirements Analysis. The IEEE defines requirements analysis as "the process of studying user needs to arrive at a definition of system or software requirements" [IEEE, 1983:30]. Software requirements analysis is therefore concerned with studying user needs to be able to define software requirements.

Valusek makes a useful distinction between requirements analysis and requirements determination. Requirements determination is a user-oriented process of developing a list of candidate requirements. Requirements analysis is the later process of focusing and reconciling these possibly conflicting requirements, and detailing them in a specification [Valusek and Fryback, 1987:147].

Land identifies four common techniques of identifying requirements. An analyst may interview users, carry out surveys, observe the system or organization in operation, or study documentation of the current system [Land, et al., 1987:208]. Prototyping can also be included in this incomplete list of methods used to identify the requirements of a software system [Gomaa and Scott, 1981].

The output of the software requirements analysis activity is a model of the problem space. This model serves a number of purposes. The model

- specifies the logical requirements without detailing a physical implementation [Gane and Sarson, 1982:9].

- expresses preferences and trade-offs of potential approaches [Gane and Sarson, 1982:9].
- focuses attention on important features of the system while de-emphasizing less important features [Yourdon, 1989:65].
- presents a basis for discussion with the user about changes or corrections to the system [Yourdon, 1989:65].
- verifies that the analyst correctly understands the user's problem [Yourdon, 1989:65].
- documents the system so that designers and programmers can build it [Yourdon, 1989:65].

2.2.2 Information Captured During Analysis. The literature identifies a multitude of elements to include in a model of the software requirements. These elements are summarized below.

- *Information Domain.* This consists of the flow, content, and structure of data. The information flow describes the manner in which data changes as it flows throughout the system. The information content represents the individual data items in the system. Information structure describes how these data items are grouped into more complex data structures [Yourdon, 1989, Pressman, 1987:142].
- *Functional Elements.* A description of the functions the system is to perform is included [Yourdon, 1989, Yadav, et al., 1988, Land, et al., 1987, Pressman, 1987:21].
- *Interface Characteristics.* The links between the system and the outside world are identified and described [Yadav, et al., 1988, Pressman, 1987:47]. This may also include the existence and frequency of any external events that the system must respond to [Peters, 1987, McMenamin and Palmer, 1981:38].
- *Design Constraints.* Any constraints on the design of the system, including performance requirements [Yadav, et al., 1988, Pressman, 1987:21], or "metarequirements" (design decisions made up front by the user) [EVB, 1989:31].

2.2.3 Requirements Analysis Tools. Many different people use the model developed during software requirements analysis, including users, designers, coders,

project managers, and maintainers. Therefore, the model must promote comprehension and communication among these parties [Jorgensen, 1986:182]. A number of tools are used to portray the information described in the previous section in an understandable manner. Each tool focuses on a different aspect of the system. Therefore, combinations of these tools are required to fully describe the software requirements. Some of the more common and useful tools are listed below.

2.2.3.1 Data Flow Diagram. The data flow diagram (DFD) reveals the processes in a system and the data flows between them. A DFD is made up of circles representing processes, arcs portraying data flows, straight lines illustrating stores of data, and boxes depicting external sources or sinks of data [DeMarco, 1979:51]. An example data flow diagram is displayed in figure 2.2.

Data flow diagrams may represent different levels of abstraction of a system. Each of the processes in figure 2.2 may be broken down and represented with its own DFD. This concept is known as the leveling of data flow diagrams [DeMarco, 1979:72].

2.2.3.2 Data Dictionary. The data dictionary is an organized, textual listing of the data items relevant to the system, containing a precise definition of each of the items [Yourdon, 1989:189]. The data dictionary is often used to support a DFD by defining the data flows and stores identified by the diagram. The contents of a data dictionary may vary with use. Gane and Sarson [Gane and Sarson, 1982:76] include the following fields:

- A definition of the data item.
- Other related data elements.
- The range of values and meanings of values for the data element.
- The length of the element.
- Any encoding used for the data.
- Other editing information.

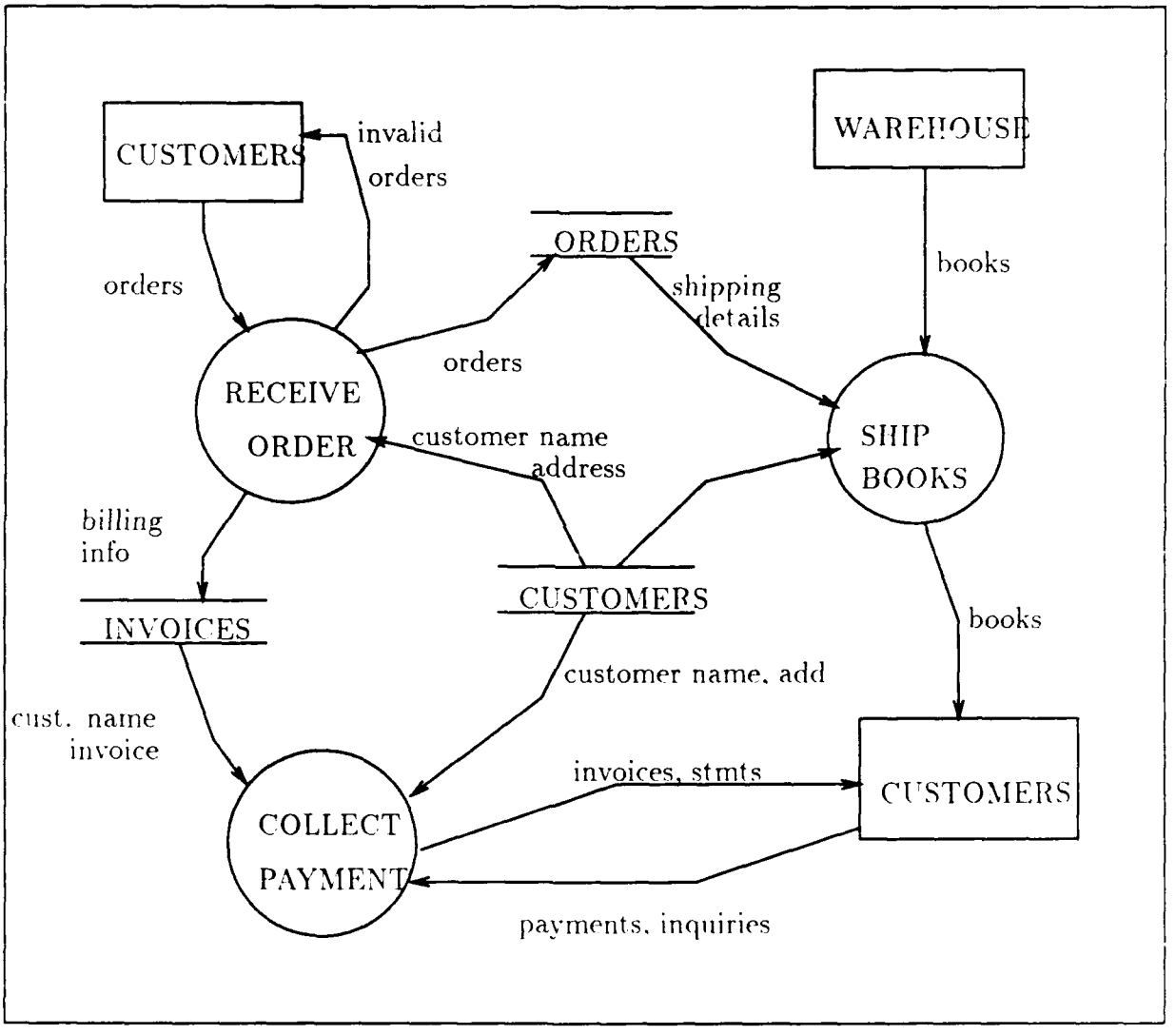


Figure 2.2. Data Flow Diagram [Yourdon, 1989:141]

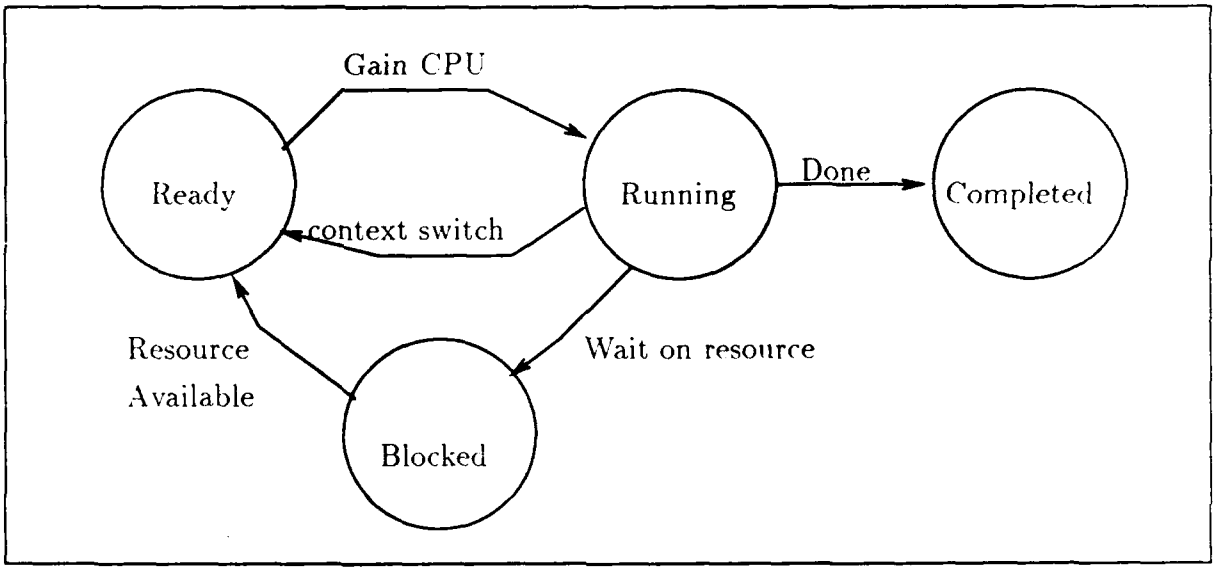


Figure 2.3. State Transition Diagram of a Process

2.2.3.3 State Transition Diagram. A state transition diagram (STD) is used to capture the time-dependant behavior of a system [Yourdon, 1989:259]. One common notation uses circles to denote each possible state of the system, and arcs to represent transitions between the states. Labels on the arcs state the condition(s) required for the state transition [EVB, 1989:163]. Figure 2.3 is an example of a state transition diagram of a process in a simple operating system.

2.2.3.4 Entity Relationship Diagram. The entity relationship diagram (ERD) was originally proposed by Chen as a tool for database design [Chen, 1976:9]. The diagram captures semantic information about the real world in terms of entities, or "things", from the real world and the relationships between them. Analysts use the diagrams to describe the layout of data stores in a system [Martin and McClure, 1985, Yourdon, 1989:233]. The basic notation of an ERD includes rectangles to denote entities and diamonds to show the relationships between them. An example ERD is shown in figure 2.4. There are various forms of entity

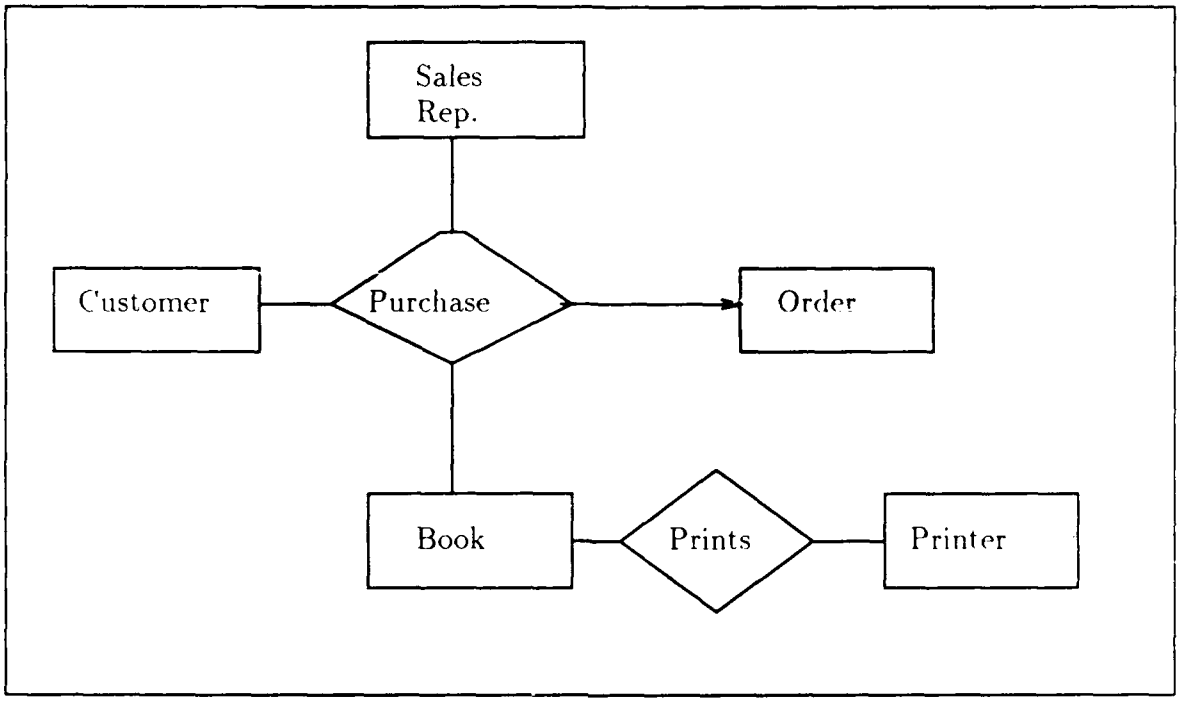


Figure 2.4. Entity Relationship Diagram (ERD) [Yourdon, 1989:235]

relationship diagrams, varying mainly in the information included about the relationships. For example, a one-to-many relationship may be shown with an arrowhead on the arc (as in figure 2.4), or with '1' and 'M' on ends of the arc [Yourdon, 1989:210].

2.2.3.5 Concept Map. The concept map is not a traditional tool for requirements analysis. Instead, it was developed by Novak and Gowin as an educational tool to summarize understanding of a topic [Novak and Gowin, 1984]. The concept map is similar to an entity relationship diagram in that it identifies important entities or concepts about a topic and describes the relationships between them. The notation used in concept maps is modest. Ovals are used to denote concepts, while labeled edges identify the relationships. This simple, unstructured notation makes the concept map easy to apply and understand.

McFarren has proposed the use of concept maps as an aid to developing decision

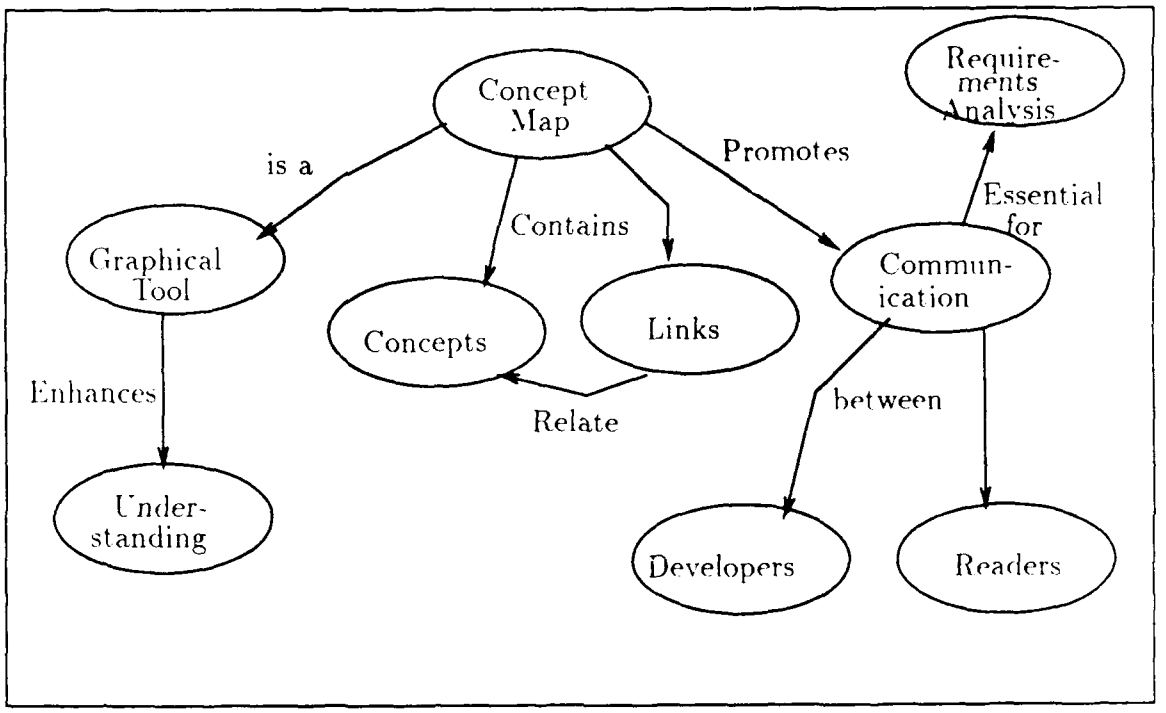


Figure 2.5. Concept Map of "Concept Maps"

support systems (DSS). The concept map helps the decision maker communicate his understanding of the problem to others, and provides a medium for identifying any misconceptions held by the DSS builders [McFarren, 1987]. Barnes [Barnes, 1988] and Umphress [Umphress, 1988] have built on this idea, proposing the use of concept maps as a tool for modeling the problem space during software requirements analysis. Barnes maintains the concept map is more descriptive than the informal strategy of OOD, allowing a more direct means of identifying the objects and operations of the problem space [Barnes, 1988:6.3]. Barnes' proposed methodology for a full life cycle object-oriented methodology, including the use of concept maps, is described in section 2.3.1.2.

Figure 2.5 is an example of a concept map describing the essence of concept maps that are important to this thesis.

2.2.3.6 Summary. The set of tools described above is by no means complete. The tools presented are those which are widely used, or, in the case of the concept map, show potential for use in the requirements analysis phase. Some combination of the tools described above can be used to represent the majority of the elements in the requirements model defined in section 2.2.2. English prose may add to or support the information presented in the diagrams. However, it is difficult to define a complex system concisely and unambiguously with a natural language alone [Martin and McClure, 1985, Yourdon, 1989, Ross and Schoman, 1977:9].

2.2.4 Approaches to Software Requirements Analysis. The selection and application of these tools depends on the approach taken toward requirements analysis. Pressman claims that all requirements analysis methods are related by similar underlying principles. In any analysis method, the following activities occur:

- The information and functional domains of the problem must be represented and understood by the analyst.
- The problem is partitioned such that detail is uncovered in a layered fashion.
- Logical and physical models of the system are developed. [Pressman, 1987:141-142]

Requirements analysis methods differ mainly in the way the problem is partitioned. Systems may be expressed in terms of data flow between system functions, data structures, events and responses, or objects. While specific analysis methods may use multiple tools to capture different views of the system, one of the views typically predominates over the others. Methods can therefore be categorized by their approach to requirements analysis.

2.2.4.1 Data Flow Oriented Analysis. Not surprisingly, requirements analysis based on data flow makes heavy use of the data flow diagram. The system

is broken up into the major functions required of the software. Each of these major functions is, in turn, broken down into its subfunctions until the system is expressed in terms of primitive processes.

DeMarco's Structured Analysis is an example of a data flow oriented analysis method. The major steps in his method are:

1) Study the current physical environment and document it in a Current Physical Data Flow Diagram.
2) Derive the logical equivalent of the current environment, and develop a Current Logical Data Flow Diagram.
3) Derive the new logical environment, as portrayed in the New Logical Data Flow Diagram plus supporting documentation.
4) Determine physical characteristics of the new environment, and produce a set of tentative New Physical Data Flow Diagrams.
5) Quantify cost and schedule data associated with each of the possibilities represented by the set of New Physical Data Flow Diagrams.
6) Select one option, represented by one New Physical Data Flow Diagram.
7) Package the New Physical Data Flow Diagram and supporting documents into the Structured Specification. [DeMarco, 1979:27]

DeMarco's approach has come under criticism recently for the amount of time spent documenting the old system. This realization has caused a trend toward starting with a model of the proposed system and eliminating the formal documentation of the existing system [Coad, 1988, Yourdon, 1989:125].

2.2.4.2 Event-Response Based Analysis. The event-response approach is characterized by the identification of external stimuli to which the system must respond [Coad, 1988, Shlaer and Mellor, 1988]. These events are used as a starting point for organizing the analysis of the system.

The event-response approach results in the development of the "essential model" of the system [McMenamin and Palmer, 1984, Yourdon, 1989]. According

to Yourdon, the essential model is composed of an environment model which defines the boundary between the system and the outside world, and a behavior model which provides a view of the insides of the system [Yourdon, 1989:326].

Development of the environment model consists of defining a statement of purpose, a context diagram, and an event list. The statement of purpose is a short (single paragraph) description of the purpose of the system, aimed mainly toward management. The context diagram views the system as a single bubble in a data flow diagram, and documents its connections to the outside world. The event list is a list of externally generated signals to which the system must respond [Yourdon, 1989:337-341].

After the environment model is defined, attention turns to the development of the behavior model. At this point, a data flow diagram is developed based on the external events identified in the environment model. A bubble is drawn for each event in the event list, and named with the corresponding response to that event. Inputs, outputs, and data stores are drawn as needed to represent communication between the bubbles. This first-cut DFD is then layered both up and down; bubbles are combined to arrive at a DFD at a level above the first cut DFD, and exploded to develop lower level DFDs. An entity relationship diagram is usually developed in parallel to document the information structure of the system [Yourdon, 1989:360-365].

2.2.4.3 Data Structure Oriented Analysis. As its name implies, data structure oriented analysis specifies software requirements by focusing on the data structure of a problem instead of the data flow. The system is therefore modeled according to an information structure of the problem [Pressman, 1987:209].

Data structure approaches share a number of characteristics. First, key data items and processes are identified. Second, the structure of information is assumed to be hierarchical. Third, data structures are represented as either a sequence of

data items, a repeated grouping of data items, or as a selection from among a set of data items. Finally, a set of steps are defined for mapping the hierarchical data structure into the structure of the program [Pressman, 1987:172].

2.2.4.4 Object-Oriented Analysis. The final approach to software requirements analysis is the object-oriented approach. This is the newest and least defined approach to analysis. There is general agreement in the sense that the overall goal is to identify objects from the real world in terms of the data and operations that compose them. The manner in which a system model should be organized in terms of these objects is still open to debate. Some specific approaches are discussed in section 2.3.

2.2.5 Summary. This section has identified some of the current tools and approaches used in the analysis of software requirements. Each approach models, in some fashion, the information domain, functional elements, and interface characteristics of the problem. The approaches differ in their choice of the characteristic(s) to serve as the basis for the model of the system.

2.3 Object-Oriented System Models

There are currently two different approaches to the development of object-oriented models of software systems. The first approach applies either data flow, event-response, or data structure oriented requirements analysis methods to model the system. This model is then transformed into a specification which models the system in terms of objects and operations. The alternative approach is to replace the more traditional approaches to analysis with an object-oriented strategy from the beginning [EVB, 1989, Ward, 1989:74].

2.3.1 Translating Traditional Models. Ward claims that there is no fundamental opposition between certain function-oriented analysis techniques (with ex-

tensions) and object-oriented design [Ward, 1989:82]. Ladden agrees that at least some of the principles of the two methods are complementary [Ladden, 1989:78], and suggests that the major difference is in the order of applying certain analysis activities. The traditional approach to analysis is to first define the functional elements and then "package", or group, similar functior . .ogether. An object-oriented approach first identifies the packages, or objects, and then identifies the functional elements associated with each object [Ladden, 1989:82].

2.3.1.1 Abstraction Analysis. Seidewitz and Stark have proposed a method for translating a data flow oriented requirements specification into an object-oriented design [Seidewitz and Stark, 1986, Seidewitz and Stark, 1987]. Their method, which they term *abstraction analysis*, uses data flow diagrams as a basis for identifying abstract entities and an initial control hierarchy. Objects, operations, and a virtual machine hierarchy are then identified [Seidewitz and Stark, 1986:5.1]. The steps involved in transform analysis, taken from [Seidewitz and Stark, 1987], are

1) Identify the central entity from the data flow diagram. This central entity is the best abstraction of what the system will do.
2) Moving away from the central entity along data flows on the DFD, identify the entities that directly support the central entity.
3) Construct an entity graph depicting the flow of control between entities. The entity graph shows the interconnection of abstract entities in the problem domain from a control point of view. The graph serves as the basis for identifying objects.
4) Develop an object diagram from the entity graph. The object diagram is based on the central entities and objects of the entity graph. The diagram (see figure 2.6) delineates objects and their required access of other objects.
5) Identify operations provided to and used by the objects.
6) Repeat the above process on lower level DFDs. This will identify subordinate objects to those already identified.

7) Translate the object diagrams into an object-oriented design in Ada. [Seidewitz and Stark, 1987:4.60-4.64].

Ladden has identified some difficulties in identifying objects from DFDs. When DFDs are used together with traditional structured design, there is usually a one-to-one mapping between bubbles on the DFD and software modules at the higher levels of design [Ladden, 1989:84]. However, when identifying objects from DFDs, the relationship between process bubbles and objects may not be as evident. Objects may overlap more than one DFD, more than one object may be identified from a single level of a DFD, and even single bubbles of the DFD may be allocated to more than one object. Another difficulty is in associating the data stores, flows and processes of a DFD with objects. This may require either grouping a number of DFDs together, or redrawing the DFDs with redundant components [Ladden, 1989:80]. According to Kenth, Seidewitz himself has admitted that it is difficult to get an object-oriented design from a specification constructed without consideration of the object-oriented paradigm [Kenth, et al., 1987:11].

2.3.1.2 Alternative Methods. One suggested alternative method of identifying the objects in the model is to supplement DFDs with other tools, such as the entity relationship diagram (ERD). Seidewitz, Ladden, and Ward all suggest the use of the ERD as a means of identifying the objects for the model. The information from the ERD is supplemented with a stimulus-response analysis [Ward, 1989:79] or DFDs [Kenth, et al., 1987, Ladden, 1989:81] to define a more complete system model.

A method defined by Barnes uses the concept map to organize the information contained in the models of traditional analysis methods. Barnes proposes the development of multiple concept maps from the requirements specification and user interviews. These concept maps are then synthesized into a single sentence statement of the problem, and a single concept map depicting a solution strategy. This

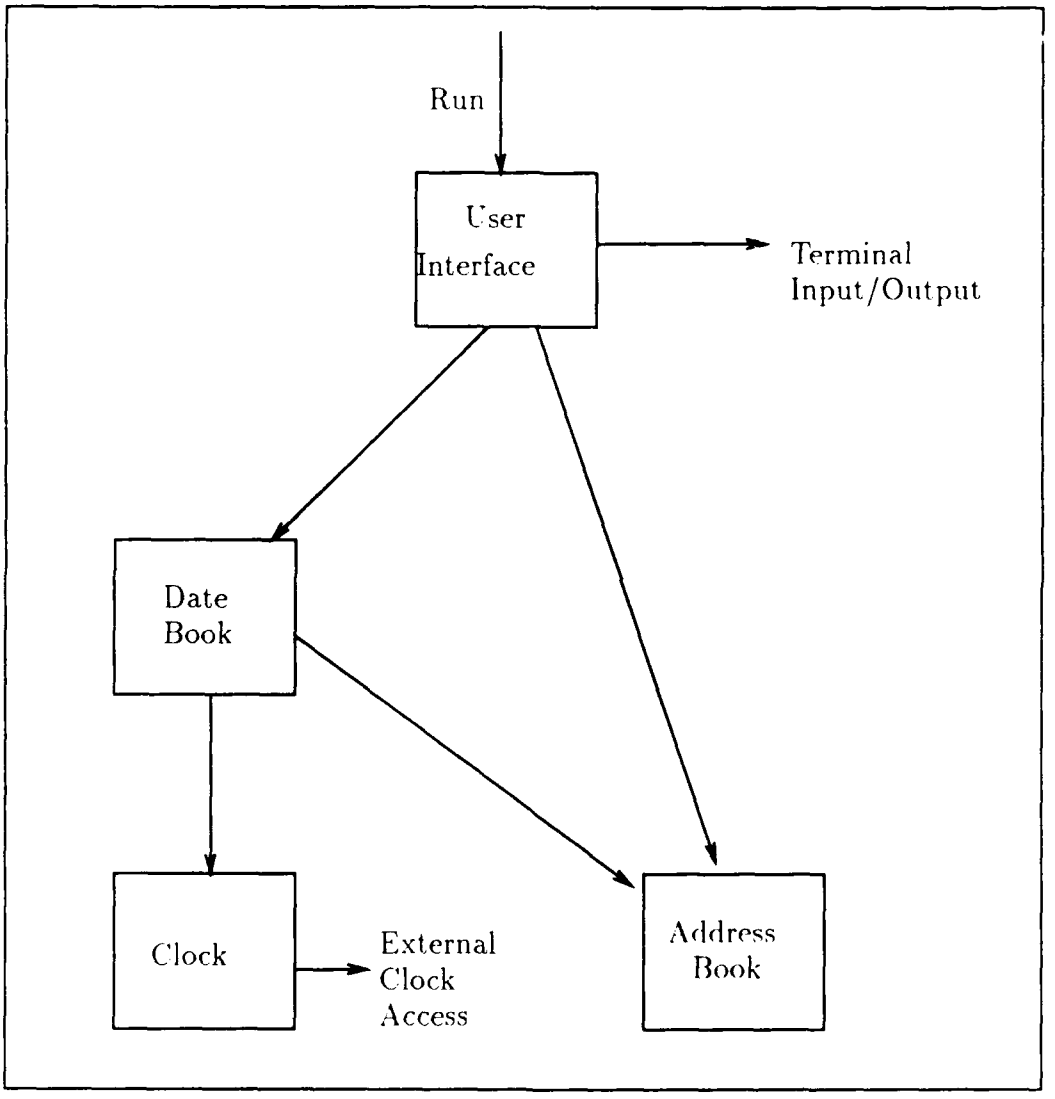


Figure 2.6. Object Diagram of a Desk Top Schedule Organizer [Seidewitz and Stark, 1987:4.56]

single concept map serves as the basis for identifying the objects, attributes, and operations needed in the solution [Barnes, 1988:3.6].

2.3.2 "True" Object-Oriented Approaches. The alternative to translating a function-oriented specification into an object-oriented design is to use object-oriented techniques from the outset of requirements analysis. The benefits of the object-oriented paradigm may be magnified with their earlier application [EVB, 1989].

2.3.2.1 Coad's Framework for Object-Oriented Requirements Analysis. As stated in section 2.2.4.4, the object-oriented approach to software requirements analysis has not reached a consensus in its specific steps. One framework proposed by Coad is to represent the system in terms of object, attribute, and process layers. The object layer identifies potential objects and their relationships. The attribute layer defines descriptive and identification attributes about the objects. The process layer defines responses of each object to external events, and the data flows between objects [Coad, 1988]. The example in figure 2.7 shows the relationships between objects at these layers.

2.3.2.2 Shlaer and Mellor's Object-Oriented Domain Analysis. Shlaer and Mellor have recently proposed an approach to object-oriented analysis based on information, state, and process models [Shlaer and Mellor, 1989:66]. Together, these models represent the system requirements. The general elements in their approach are:

1. *Information Models.* A detailed version of Chen's entity relationship diagram is used to identify the objects, attributes and relationships of the problem. (See the portion of an information model of a juice factory in figure 2.8.
2. *State Models.* The life cycles of the objects are expressed using state transition diagrams.

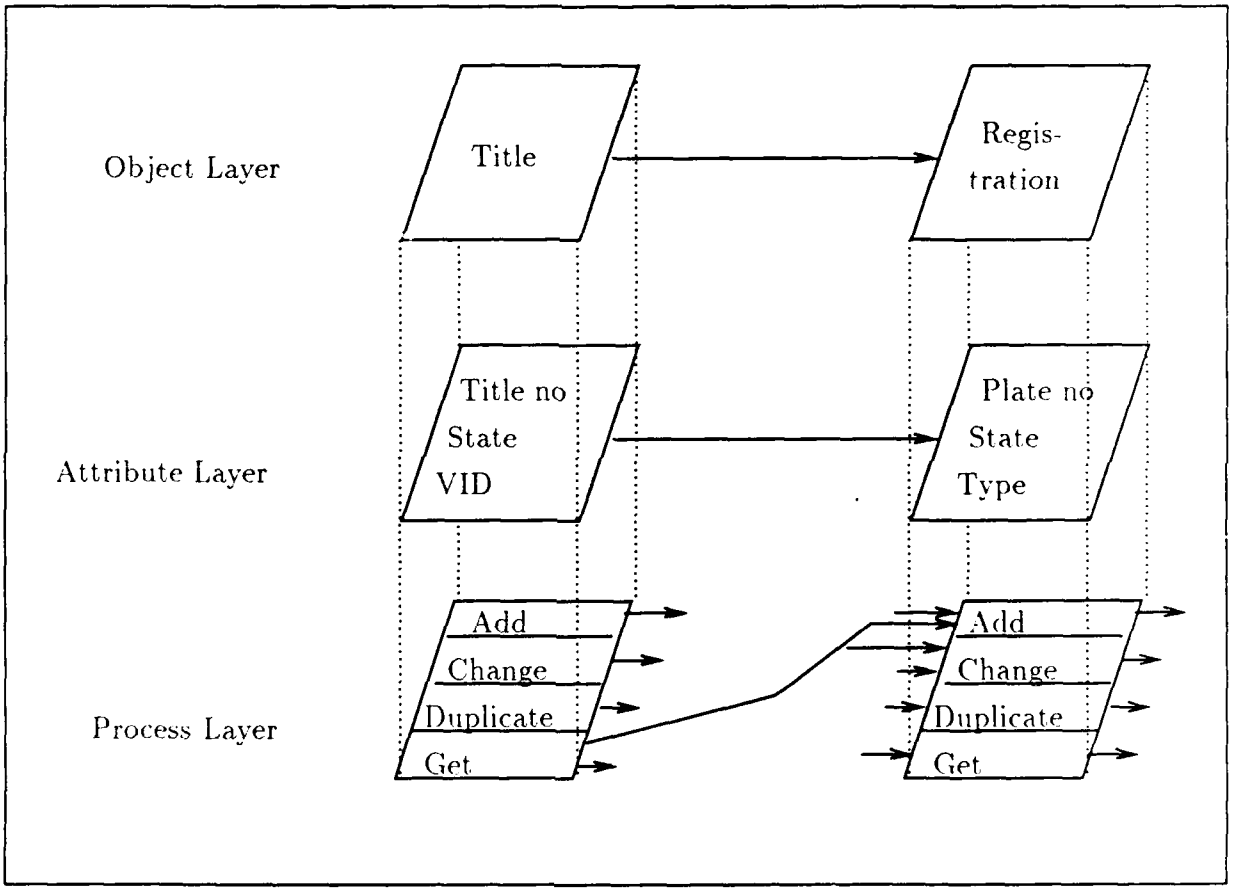


Figure 2.7. Coad's Object-Oriented Framework [Coad, 1988]

3. *Process Models.* The state transition diagram is used to identify the processes required to drive an object through its life cycle. Data flow diagrams are used to depict the action processes for each state in the state model.
4. *Boundary Statement.* The external boundary of the automated portion of the system is identified.

Shlaer and Mellor recognize the relationship between the nature of an object (as actor, agent, or server) and it's location in the system hierarchy. Those objects at the upper level of abstraction tend to be actors sending messages to guide lower level objects through their life cycles. Objects in the middle levels are usually agents, receiving messages from the upper level objects, and requesting operations from the lowest level objects. The objects at the lowest level of abstraction are often unintelligent servers, typically used to directly model a hardware entity [Shlaer and Mellor, 1989:74–75].

2.3.2.3 Bailin's Object-Oriented Requirements Specification Method. Sidney Bailin proposes another new method of transforming a textual requirements statement into a more formal, graphical model. His method uses both a set of entity relationship diagrams and a hierarchy of entity data flow diagrams (EDFDs) to capture the system requirements. An EDFD is similar to a traditional DFD, except that the nodes may be entities as well as functions [Bailin, 1989:609]. Each function is performed in the context of some entity.

The steps in producing the specification are described below:

1. *Identify key problem-domain entities.* An entity relationship diagram is used to record the problem domain entities and their interrelationships.
2. *Distinguish between active and passive entities.* Intuitively, active entities act as processes, while passive entities are data flows. Bailin revises this definition to consider active entities as those whose functions are important to consider during requirements analysis. A

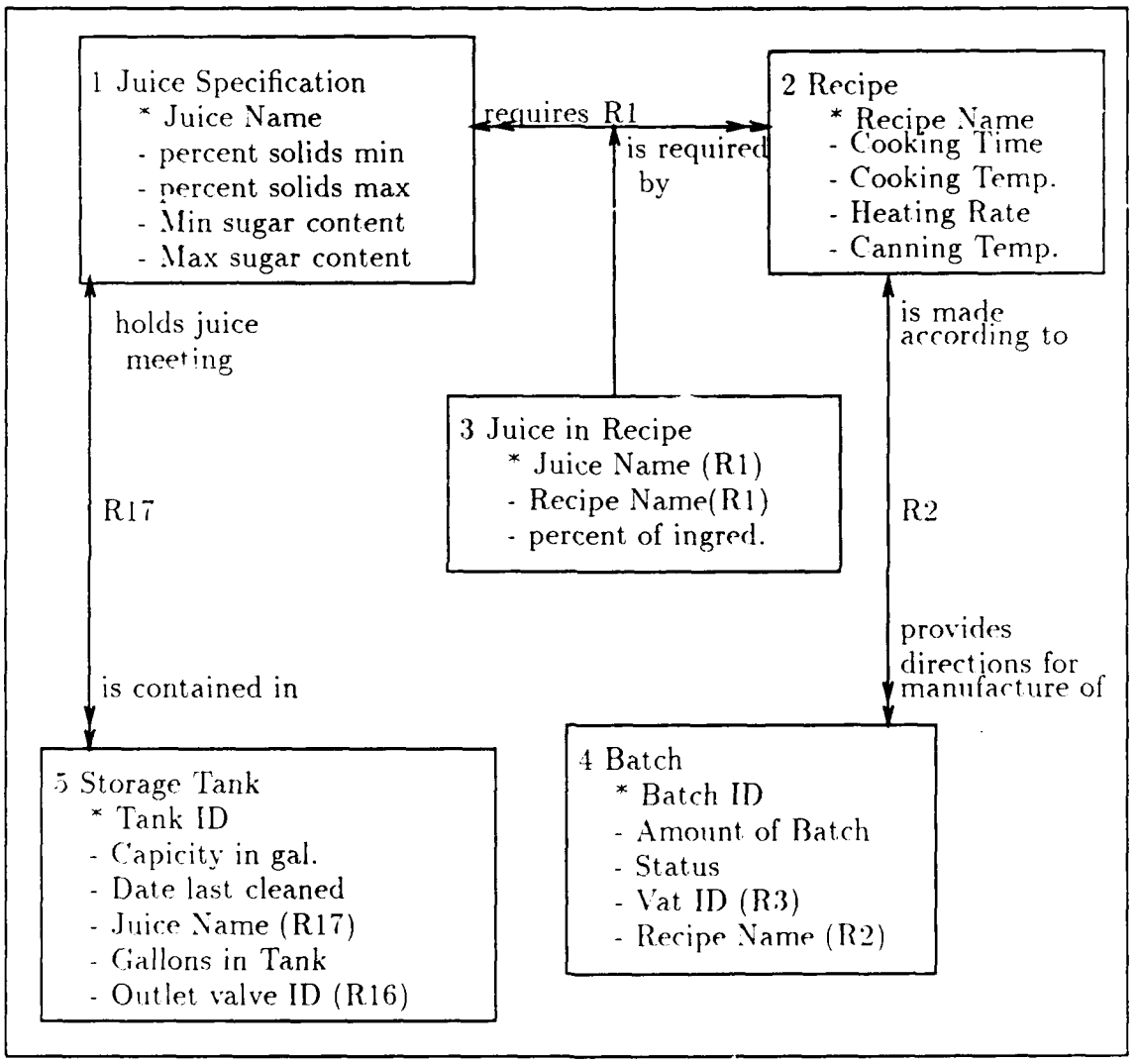


Figure 2.8. Shlaer and Mellor's Information Model of a Juice Factory (partial) [Shlaer and Mellor, 1989:69]

passive entity is then one whose functions need not be considered until the design phase.

3. *Establish data flow between active entities.* Entity data flow diagrams document this flow. Each active entity in the ERD becomes a process in the EDFD. Passive entities appear as data stores, or as flows between entities.
4. *Decompose entities (or functions) into sub-entities and/or functions.* These next three steps form the heart of the method. In this step, the EDFD is decomposed into subentities and/or functions in a new EDFD. Subentities *compose* the entity, while a function is *performed by* the entity.
5. *Check for new entities.* The new EDFDs are scanned to see if they imply the existence of new entities. These new entities, if significant, should then be included in the entity relationship model.
6. *Group functions under new entities.* For each of the new entities introduced in the previous step, the functions performed by or on the new entities are identified. Existing functions may be rearranged to fall under one of the new entities. The goal of this step is to identify the functions to ensure that the entity is functionally complete.
7. *Assign new entities to appropriate domains.* Finally, the new entities are assigned to some hierarchical domain. The entity relationship diagram, if complex, can be redrawn to reflect this hierarchy.

2.3.2.4 EVB's Object-Oriented Requirements Specification. A final method of object-oriented requirements analysis is suggested by EVB Software Engineering, Inc. The requirements analysis process is documented in an object-oriented requirements specification (OORS) which is divided into an object-general section and an application-specific section [EVB, 1989:123].

Object-General Section. The object-general section of the OORS contains an object and class specification (OCS) for potentially reusable objects and classes required for the problem. The OCS consists of the following elements:

- A textual description of the object or class

- Graphical representations of the static and dynamic characteristics of the object or class. The static relationships of the object or class to other objects or classes is captured in a semantic net or entity relationship diagram. The dynamic behavior of the object or class is represented in a state transition diagram, or, for complex behaviors, in a petri net graph.
- A list of operations suffered by the object, or operations the object or class requires of other objects.
- Documentation of the state information of the object or class, including restrictions on the state of the object.
- A description of any constants or exceptions applicable for a class. [EVB, 1989:123]

Application Specific Section. The application specific section of the OORS documents the elements of the system specific to the problem at hand. This section contains four divisions. The first contains the OCSs for the application specific system components. The second section consists of OCSs for any components specified by a design decision made by the user (a metarequirement). The third segment lists any qualifications on components based on how or where they are used in this system. The final division of the applications specific section is a "precise and concise" description of how the objects and classes interact in the system to solve the problem. This description ties together the elements of the software system by describing items such as the user interface, timing constraints, system limitations, etc. [EVB, 1989:241–255]

Together, the object general and application specific sections form the object-oriented requirements specification. This OORS is the basis for developing an object-oriented design.

2.4 Summary

The widespread use of different approaches to software development suggests that there is no single "right" way to apply software engineering principles. None of the proponents of the various approaches claim that their method is universally

applicable; however, few concrete guidelines exist for determining which approach to apply to a particular problem.

Regardless of the approach used for software development, it seems customary to apply that strategy in both the analysis and design phases. If a data structure oriented approach is used to uncover detail in the analysis phase, the same tactics are normally applied during the design phase as a basis for defining the architectural structure of the software. Likewise, if software is decomposed based on the functional elements of the system, activities in both the analysis and design phases are aimed towards specifying and constructing these functional elements. In either case, the model produced during the analysis phase maps naturally into the design phase.

The use of object-oriented techniques in the design phase requires some preliminary effort to identify the objects required for the solution. The application of traditional analysis methods results in a functional or data structure oriented model of the problem space. These models do not map as naturally into an object-oriented design, requiring some sort of translation into an object-oriented model of the requirements prior to design. This translation may be difficult and obscure.

The literature points to a trend in applying object-oriented techniques from the inception of the project. Though immature, these techniques show promise in developing models of software requirements that have a more natural mapping into an object oriented design.

III. An Object Oriented Analysis Method

The last chapter discussed the state of the practice in applying object-oriented design. My experience in teaching Ada and OOD agrees with those, such as [Ladden, 1989], who reject the informal strategy as the basis for constructing an object-oriented design. Students of OOD find it difficult to come up with an informal strategy which is both complete and descriptive of the problem. They seem to be obsessed with the syntax of the English paragraph instead of the meaning it is supposed to portray. Even experienced designers find it difficult to come up with an informal strategy without working backwards from a more intuitive attempt at the design.

The practice of using traditional analysis tools (e. g. DFDs) to specify the problem [Booch, 1986, Seidewitz and Stark, 1986] is a step in the right direction. However, as discussed in the previous chapter, this approach also has problems. Often, there is not a clean, one-to-one mapping between the bubbles on a data flow diagram and the objects or operations in the system. Thus, the transformation from these tools to an object-oriented design is confusing and difficult. Also, a supposed benefit of an object-oriented representation is that it more closely matches the structure of the real world problem. It therefore seems to make little sense to first model the problem using function-oriented tools and then translate the model into an object-oriented representation.

The objective of this chapter is to present a method of modeling software requirements with an object-oriented approach from the outset. The chapter first outlines the requirements for the object-oriented analysis (OOA) method. Next, it describes the general approach of the method, and presents a detailed discussion of the method steps. Finally, it discusses the mapping of this model into an object-oriented design.

3.1 *Goals of an Object-Oriented Analysis Method*

The following guidelines should be considered in the formation of an object-oriented analysis method.

3.1.1 User Orientation. The first objective of an object-oriented analysis method is that it be "user friendly". In other words, the models developed under the method should be developed with the user, or domain expert, in mind. Too often, analysis tools (with their cryptic syntax) are aimed at the design end of the life cycle, leading the analyst off down a dangerous path. As Roland Mittermeir put it:

> Both user and analyst are very soon involved in too much technical detail to recognize they are travelling very well on a good road, but the road may lead in the wrong direction. Users cannot discover this mistake, because the symbols that are shown on the analyst's road map do not sufficiently relate to them, and the analyst cannot see it either, because he lacks knowledge about the detailed environment. [Mittermeir, et al., 1987:154]

The tools of the method should be primarily graphical, with supporting textual information. The tools should also require minimal instruction, so that domain experts can quickly learn to develop or critique the software models. The notation of the tools should be consistent whenever possible.

3.1.2 Ease of Use. Likewise, the analysis method should be fairly easy to apply by an analyst. The complexities of the requirements of a large software system will tax the analyst enough without the added difficulties of applying a labyrinthine set of steps and tools. Barnes makes the interesting observation that the amount of use enjoyed by a particular method is inversely proportional to its complexity [Barnes, 1988:2.34].

3.1.3 Information Captured. Section 2.2.2 identifies some general elements of the problem that are captured in a requirements analysis method. These elements included the interface characteristics, information domain, functional elements, and design constraints. These problem aspects should be captured in the OOA method. In addition, an object-oriented analysis method is specifically concerned with identifying the objects in the problem, defining the attributes of these objects, and recognizing the active relationships, or messages passed among these objects.

3.1.4 Other Requirements. In addition to the above goals, the object-oriented analysis method should:

- model the system in a top-down hierarchical manner. The details of the problem should be presented in layers of abstraction, beginning with the most general concepts.
- support the definition of embedded systems requirements. After all, this is the stated application domain of Ada.
- support requirements analysis of large software systems. The tools and guidelines should consider the complexities of large systems. Tools (such as the "informal strategy") which are useful only for describing small problems have limited use in modelling large software systems.
- include minimal redundancy. Redundant information is useful in checking the consistency between multiple views of the problem. However, redundancy also makes it difficult to update a model when that information changes. The OOA method should emphasize modifiability over redundancy.
- map into OOD. The output of the analysis method should map cleanly into the design phase, where a Booch-flavored Ada object-oriented design is carried out.

The general goals stated above guided the selection of tools and steps which make up the object-oriented analysis method of this thesis.

3.2 General Approach to Object-Oriented Analysis

The ideas which make up the object-oriented analysis method were synthesized from many sources. The method was greatly influenced by the works of [Booch, 1986], [Yourdon, 1989], [EVB, 1989], and others. The influence of EVB's object class specification (OCS) is particularly evident in the method's description of each class of objects. However, despite some similarities in the form of class documentation, this OOA method is clearly different from EVB's in the method steps and tools used to identify the objects and operations of the problem.

3.2.1 Role in the Life Cycle. The generic view of the software life cycle consists of three phases: definition, development, and maintenance [Pressman, 1987:27]. The OOA method addresses the software requirements analysis activity in the definition phase. The method assumes that a *systems* analysis has already been done to define the hardware-software boundary.

If there are areas of uncertainty in the software requirements, then a series of prototypes may be justified to better understand these areas. Tools in the OOA method may help to document desired modifications to the prototypes. At some point it will be possible to identify the requirements for a major release. The OOA method can then be used to document this baseline set of requirements in a specification. This object-oriented requirements specification will be useful later during development and (especially) maintenance of the software.

The OOA method does not attempt to identify all potential objects that will be present in the final design. The method will only identify those objects which are evident from the definition of the problem and software interface. The designer should expect to identify additional objects and operations during the development phase that are required for the complete solution. The distinction between analysis and design is a fine line--an attribute of an object is an object in its own right at the next lower level of abstraction. The analyst should document objects and classes

to the level of abstraction where the *domain expert* is confident that the essence of the problem is captured.

3.2.2 Method Tools. The object-oriented analysis method was conceived by first identifying the tools and models required to satisfy the goals listed in section 3.1. Once the end products of the OOA model were selected, the steps in constructing this model were defined.

The OOA method attempts to bridge the gap between the problem domain expert and the designer. The nature of the communication with these parties is different. The domain expert's view of the world often lacks the structure desired by the designer. Therefore, the analysis method must transform an unstructured view of the problem into one which is structured enough to minimize uncertainty and ambiguity. The method steps are guidelines for this transformation from the domain expert's view to the designer's view. These steps are not automatic—they require the intuitive judgement of the analyst and review of the domain expert to fill in any gaps in the representation of the software requirements.

3.2.2.1 Communication with the Domain Expert. In the OOA method, communication with the domain expert is handled primarily through concept maps, story boards, and a list of external events and desired responses. As described in section 2.2.3.5, the concept map is an unstructured entity relationship diagram. The unstructured nature of the concept map enables the domain expert to draw and understand it with minimal training. This set of concept maps communicates a general understanding of the problem elements to the analyst and designer. The maps are also used to identify the objects describing the problem and their attributes.

The event/response list identifies external stimuli to which the software must respond. The story boards provide a means of depicting scenarios from which the events and responses are identified. The events will later be viewed as messages that need to be passed between objects, in the form of an object calling upon an operation

provided by another class of objects. The response of an object to a message may also imply additional messages that the object must send to other objects in the system. Together, the concept maps and event/response list paint an insightful picture of the software requirements.

3.2.2.2 Communication with the Designer. The information from the domain expert's concept maps, story boards, and event/response list is conveyed to the designer through a set of entries in an "object encyclopedia". This encyclopedia is similar in concept to a data dictionary, but its entries contain more comprehensive information than a traditional data dictionary. The major components of such an entry are:

- A *textual description* of the object or class.
- An *interface diagram* showing the messages an object or class receives and passes to other objects.
- A *structure diagram* illustrating the sub-objects or attributes of a class of objects.
- A *state transition diagram* displaying the states of an object and the transitions among them.

These items are described in more detail in section 3.3.2.6.

3.3 Steps in the Object-Oriented Analysis Method

The Object-Oriented Analysis (OOA) method consists of the following steps:

1. Capture the domain expert's view of the software. This is accomplished through the following actions:

a) Define the overall purpose of the software.

b) Draw a set of general concept maps which describe the overall problem.

c) Outline any user interface and operational scenarios with story boards.

d) Produce an event/response list for the software.

e) Identify known restrictions on the size, reliability, or execution time constraints of the software.

f) Identify any domain expert imposed design decisions ("metarequirements") for the software.

2. Model the software requirements in a top-down, hierarchical manner. In this phase, the following guidelines apply:

a) Draw an external interface diagram for the software component.

b) Identify any high level actor objects which perform some overall algorithm.

c) Construct a preliminary object list.

d) Identify the senders and receivers of the messages/events.

e) Document the object classes.

These steps are covered in more detail in the following paragraphs.

3.3.1 Step One: Capture the Domain Expert's View. The first step of the object-oriented analysis method aims at capturing the domain expert's view of the problem. Step One may be performed either by an analyst working with one or more domain experts, or by the domain experts themselves. The emphasis in this step is in conveying understanding of the problem from the domain expert(s) to the analyst. At this point, little structure is imposed on the information captured.

3.3.1.1 Step 1a: Define the overall purpose of the software. The statement of purpose simply gives the reader a starting point for understanding the requirements of the proposed software system. The length of this description may vary with the complexity of the system, but can be as short as a single sentence. To emphasize only the essential elements of the problem, the upper limit should be one page.

3.3.1.2 Step 1b: Draw general concept maps of the problem. These concept maps are to provide a general understanding of the *elements* of the overall problem. At this point, no structure is imposed on the format of the concept maps. The domain expert is free to lay out the problem as he sees fit. The maps will later serve as the basis for identifying the objects of the problem space and their characteristics.

There are a number of sources of input into concept maps. The analyst may draw upon a textual statement of preliminary requirements, observe the design or operation of a previous system, conduct surveys, or interview domain experts. When developing concept maps from interviews, it is important for the analyst to do some "homework" before the interview so he has some idea of the important aspects of the problem. Initial concept maps may be redrawn later to clean them up.

When drawing a set of concept maps, the analyst should keep in mind the central concept of each particular map. The maps should identify both static (structural) and dynamic relationships between the entities. The maps should concentrate on the problem aspects that are important to the software solution—it should not emphasize physical details (e.g. color, physical location) that are not important to the solution of the problem. Likewise, a single map should not be packed with too much detail. If the concept map does not fit *cleanly* on a single page, attempt to move the "peripheral" concepts and/or relationships to more detailed maps and concentrate on the central concept of the individual map.

The perception of the problem by domain experts may change over time due to recent problems or situations. Also, different domain experts may have different views of the problem. Therefore, it is desirable to obtain concept maps from several experts in the application domain, and over a period of time. These maps will have common nodes, indicating the most common and consistent elements of the problem. The concept maps should then be combined into a single set of maps portraying a consolidated understanding of the problem. This set of concept maps may then be reviewed by the domain experts in an attempt to breed a consensus view of the

problem.

3.3.1.3 Step 1c: Construct story boards. Story boards serve as an early paper prototype of the proposed software. Story boards are useful in specifying the physical layout of a user interface. Screen displays and menus can be drawn in story boards, giving the domain expert and analyst a feel for the system as it plays out a number of situations through different story boards. However, story boards are not limited to portraying only the physical layout of display screens. Story boards can be annotated with logical, as well as physical, entities depicting the state of some object as it responds to external stimuli.

These models for the interaction of the software with the environment are useful to prototype the "look and feel" of the software at an early point in the life cycle. A series of story boards can assist in capturing a sequence of interactions between the software and the environment, much like a comic strip tells a story through its sequence of frames.

This sequence of actions portrayed through the story boards is useful in acting out scenarios the software may face. The scenarios will be useful in later steps to identify external events and responses, and to construct state transition diagrams for object classes.

3.3.1.4 Step 1d: Produce an event/response list for the software system. The event/response list provides an action-oriented view of the problem to complement the more structural view portrayed in the concept maps. This list will include all events external to the software to which it must respond. The event/response list includes a short description of the response to each event, as well as any information concerning the frequency and volume of the event and any maximum response time. If the event is periodic in nature, the analyst should note this fact.

When developing the event/response list, it may be helpful for the analyst

and domain expert to walk through different scenarios from the perspective of the software system. These scenarios, acted out through the aid of the story boards, may aid in the identification of events to which the software must respond. The events and responses evident from these scenarios form the heart of the list. Some of the arcs on the domain expert's concept map which are labeled with action verbs are also potential candidates for these events. The events in the list may be initiated either periodically, or due to some stimuli from an entity external to the software component.

The responses to each event should be written in enough detail with respect to the problem elements. For example, if an event in a cruise control system is the pressing of the *accelerate* button, the associated response should be specific as to what needs to be done by the system. Therefore, "increment the *desired speed*" is probably better than "go faster". An event may require multiple or conditional responses. If the response is complex, it may warrant more than a simple sentence.

The analyst should cross-check the set of concept maps and the event/response list. Each object stated or implied from the nouns in the event/response list should be included in the set of concept maps previously developed. Although this may require redrawing the concept maps, it ensures that the concept maps adequately address all phases of the problem. The event/response list and concept maps may be developed concurrently.

The event/response list will be used in phase two of the method to aid in identifying the messages passed between objects.

3.3.1.5 Step 1e: Identify any restrictions on the software. Any known physical or regulatory restrictions on the software should obviously be stated as early as possible. Such restrictions may include the size, reliability, execution time, or security of the software. Documenting this information at this time could avert a costly design error.

3.3.1.6 Step 1f: Identify any "metarequirements". Metarequirements are design decisions imposed on the system by the user (or even higher authority). For example, use of a certain internal data base format may be dictated to ensure consistency with existing or future software.

3.3.2 Step Two: Add Structure to the Requirements. The second phase of the object-oriented analysis method entails modeling the software requirements in a top down, hierarchical manner. Each class of objects, and their inter-relationships, are identified and documented in this phase of the method.

3.3.2.1 Step 2a: Draw an external interface diagram of the software component. This diagram puts the software system in context with the outside environment. The events in the event list (with the possible exception of periodic events) will come from the external entities shown in this diagram. The external entities often show up on the set of concept maps developed in the first phase of the method.

3.3.2.2 Step 2b: Identify any high level actor objects which perform some overall algorithm. When implemented in Ada, the "middle part" of the external interface diagram is often an all-encompassing actor object which sends messages to other objects to dictate the flow of control of the software. The algorithm run by this object is the "glue" which ties all of the objects together by defining a sequence to the sending of messages.

The need for such a high level actor object is hinted at in figure 3.1. The real-world objects and operations of the problem are encapsulated into software representations of the entities. Apart from this, there may be an algorithm which manipulates the objects in the problem. In Booch-flavored object-oriented design, this algorithm takes the form of a high-level actor object sending messages to the objects. (The nature of this high level algorithm may be similar to the "informal

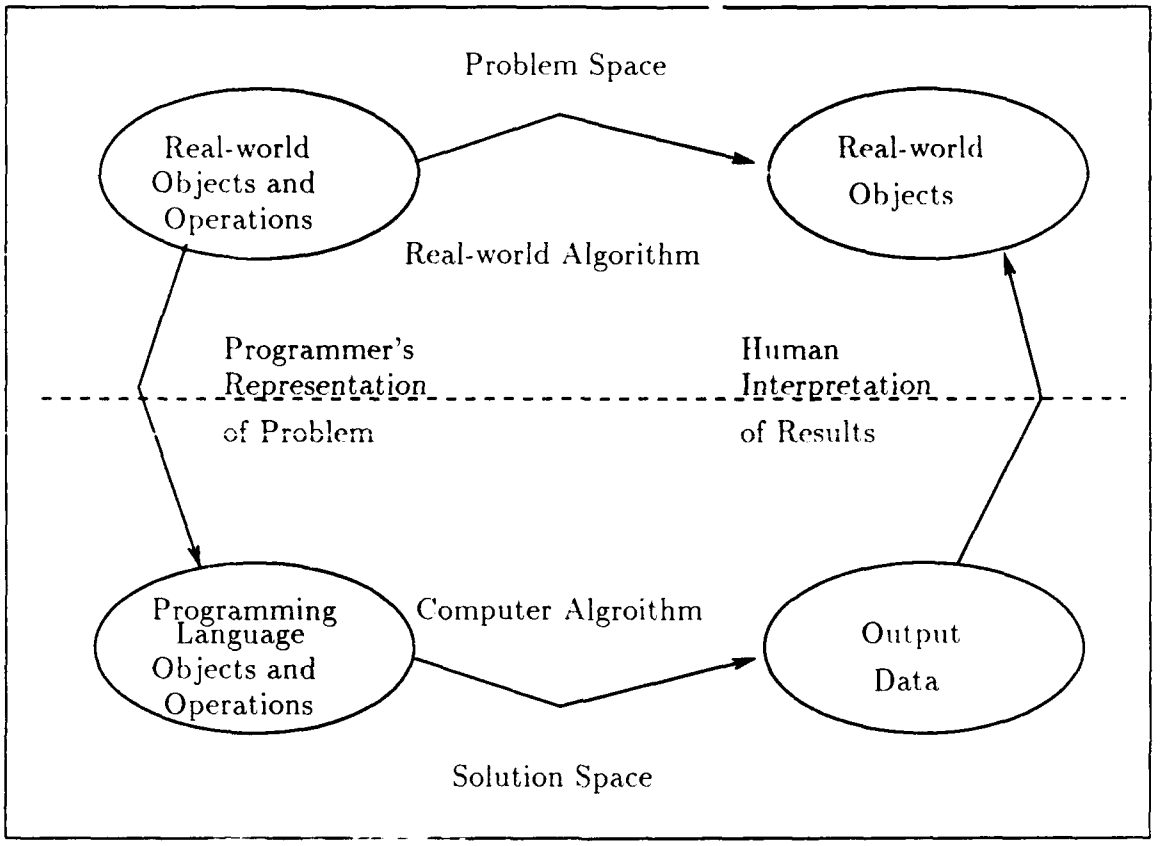


Figure 3.1. Relationship of Objects and Algorithms [Booch, 1983:39]

strategy" of Booch's initial OOD method.) This overall object is typically the "main program" in an Ada implementation of OOD. If the algorithm implemented by this overall object is complex, this object may be broken down into subobjects, implemented by Ada tasks, each implementing different logical areas of the problem. An example is the *Environment Monitoring* design problem in [Booch, 1987b].

At the highest level of abstraction, it may be difficult to distinguish between a high-level actor object and a functional process. The definition of an object presented in the last chapter requires that an object maintain some state information. However, this state may be simply the composite st. es of each of its sub-objects.

At this step in the method, the analyst should identify any high-level algorithm controlling the objects. In complex problems, this highest-level object should now be decomposed into multiple actor objects, each controlling some logical area of the problem. At this point, the analyst would then draw a structure diagram of the top-level object to illustrate this decomposition. The domain expert's concept maps and the event/response list may provide some insight into this decomposition.

3.3.2.3 Step 2c: Construct a preliminary object list. The next step of the method is to construct a working list of objects that will potentially appear in the solution. The objects will normally come from the concept names on the domain expert's concept maps. The nouns in the description of any high-level algorithm may also imply additional objects. In addition to these guidelines, the concepts of abstraction and information hiding may help divide the problem up into objects. In the spirit of [Parnas, 1972], each object should hide the implementation of some abstract entity from the problem. *Include entities from the external interface diagram* in this list.

Once a list of objects is identified, group the objects with similar characteristics under the name of a class that encompasses those objects. If there is a "large" number of object classes, attempt to group logically related object classes into subsystems. A subsystem denotes a logical collection of cooperating structures and tools [Booch, 1987a:615]. In other words, one can think of a subsystem as a set of logically related objects that forms some entity at a higher-level of abstraction. The grouping of objects should form a manageable hierarchy of subsystems and objects.

3.3.2.4 Step 2d: Identify message senders and receivers. Each of the events in the event/response list can be viewed as a message between two objects. The response corresponding to each event briefly describes the algorithm to be implemented by the receiver of the message. This step requires the analyst to identify the sender and receiver for each of these messages.

If the event has more than one response, it may be that these responses should be performed by different software objects. In this case, identify which objects perform each of the responses. The main receiver of the message will then have to forward the message to other objects to signal them of the event.

If none of the objects previously identified are appropriate as a sender or receiver of the message, this is an indication that either an external entity is missing, or that the object(s) identified thus far are not adequate. A new object may have to be added to the list. However, periodic events may have no explicit sender, unless the source is some timer object.

3.3.2.5 Step 2e: Document the object classes. Each object class is documented with an entry in the "object encyclopedia". Descriptions of external entities are included if their interface is modeled in software.

The analyst begins by drawing an interface diagram for the top-level actor object. This diagram shows the access requirements between the high-level object and the objects at a certain level of abstraction. At the top level, most of the messages from the system-level event/response list will appear as arcs between the all-encompassing object with other objects. However, it may happen that some of the messages from the system-level list may be more appropriate at a lower level of abstraction. At some point in the analysis review, the analyst should ensure that all events in the event/response list are shown as messages to an object.

For each class of objects shown on the interface diagram, the analyst enters a new reference in the "object encyclopedia". If new objects from the problem space are uncovered, they are included in the list of objects developed earlier. (Be careful not to add objects to the list that are only part of the solution and are not required to describe the problem. These objects will be identified in the design phase.)

When modeling external entities as objects, the classification of these external objects as actors or servers depends on the nature of the external device (i.e. polled

vs. interrupt driven). Some of these external entities (such as a keyboard) may be accessed through the operating system instead of implemented as a software object. In this case, documentation of these purely hardware entities may not add anything to the specification.

In some circumstances it may be helpful to more explicitly document the interaction between multiple objects. In this case, the analyst may want to include a separate Petri Net Graph depicting this complex interaction among objects.

This process is repeated at lower levels of abstraction until all objects from the object list are documented, and the software is modeled to such a level of detail where the problem is well understood. As stated in section 3.2.1, the object-oriented analysis method concentrates on defining the problem with respect to its interface with the outside world, as defined by the domain expert. To attempt to document the problem below this level seems to involve specifying more of the "how" than the "what" of the problem. The analyst should try to refrain from inadvertently crossing this fine line between analysis and design.

3.3.2.6 Contents of an "Object Encyclopedia" entry. Each class of objects in the object encyclopedia is documented with a textual description of the object, a structure diagram (showing its attributes/sub-objects), an interface diagram (showing the communication of this object to other objects in the system), a state transition diagram (if appropriate), a description of any limitations on an object's state, a characterization of messages received (operations provided), a description of messages sent (operations required) to other objects, a list of exceptional (error) conditions the object flags, a list of constants exported, a list of objects in the class being documented, and any reuse considerations for the class.

The textual description of a class of objects simply states the purpose of the object class. It also may include any miscellaneous information about the class not included anywhere else.

The structure diagram is a "pseudo" concept map. It contains concepts and relationships as in a standard concept map, but the relationships are limited to the structural relationships of the class being described. The structure diagram exposes the internal view of a class of objects, documenting its attributes or sub-objects. It can be drawn using as a guide the concepts linked by structural verbs (e. g. is a, has a, etc.) on the domain expert's concept map.

The interface diagram is also a "pseudo" concept map. It displays the external view of the class—messages sent or received by the class. When drawing an interface diagram for a class of objects, it is helpful to list the events and responses for the individual class. This list aids the analyst in identifying the messages sent and received. The events should match with the messages received by the object class, while the response descriptions (along with the action-verb links on the domain expert's concept map) will hint at the messages sent to other objects. The messages sent and received by the object are documented in the corresponding text as well. This text further describes the significance of each of the messages. In the list of messages sent, the class name of the receiving class is included, unless the class is reusable and the receiving class varies between objects in the class.

The state transition diagram (STD) for the class of objects may also aid the analyst in identifying messages that an object receives. The STD may indicate that a certain message must be received to transition into a certain state. The analyst may also study the structure diagram for the class to ascertain whether a selector, constructor, or iterator operation need be provided for each attribute. The identification of operations required and provided should be influenced by the concept of *object coupling*. It is desirable for an object to exhibit *black box coupling*, where the method for each message received requires knowledge of only the class being documented, rather than *white box coupling* where the method requires knowledge of other objects in its implementation [EVB, 1989:101]. Finally, if a message received by the class of objects involves a complex algorithm in its response, an outline of

this algorithm may be included in the message description.

The analyst draws the state transition diagram based on any state information implied in the domain expert's concept map, story boards, or event/response list. The messages received by an object may indicate a change of the object's state is required. Limitations may exist on the state of an object. These limitations may include:

- A limit on the number of items in a homogeneous composite item.
- A limit on the range of values in a scaler class.
- A limit on the length of time an object may be in a particular state.

These limitations are documented in the textual information for the class.

The series of structure and interface diagrams defines a hierarchical model of the objects in the problem space. The object encyclopedia entries may be grouped either alphabetically or hierarchically, from the highest level of abstraction to the lowest. The hierarchical grouping of entries seems best managed with a software tool so that the analyst can easily get from one level of abstraction to another. The requirements for such a tool are described in the next chapter.

Finally, the object-oriented analysis method assumes that the domain expert will be involved in reviewing the products produced by the analysis. His input and review is essential in phase one; in fact the domain expert(s) may perform this phase of the analysis independent of a separate analyst. The domain expert's review is also crucial to the success of the model of the requirements developed in the second phase of the OOA method. Although the nature of this model is more structured than the first phase, the domain expert should be able to follow the model as an extension to the concept maps and event list he provided earlier. The final model of the requirements provides the bridge between the domain expert and the designer, so both should understand and agree on the model before more formal design begins.

3.3.3 Sample Analysis Problem. The following example shows the OOA method applied to the requirements analysis of a typical cruise control system for an automobile. Enough of the analysis is presented to provide an indication of the intended use of the method tools.

3.3.3.1 Step One: Capture the Domain Expert's View. Phase One of the method entails the following steps:

Step 1a: Define the Overall Purpose of the Software.

The cruise control system adjusts the automobile's accelerator to automatically maintain a constant vehicle speed.

Step 1b: Draw a Set of General Concept Maps Which Describe the Overall Problem.

The set of concept maps is shown in figure 3.2 and figure 3.3.

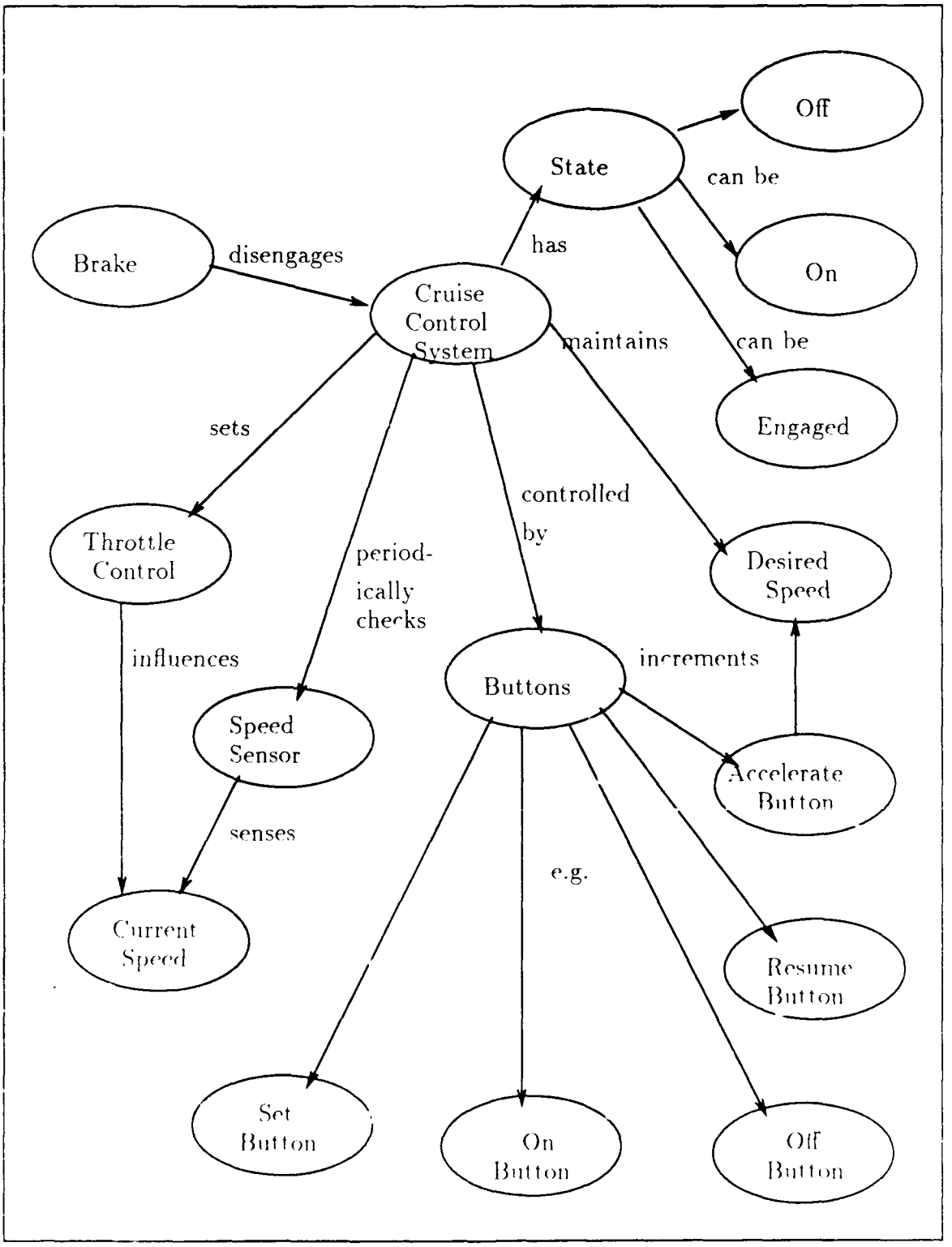


Figure 3.2. Concept Map: Cruise Control System

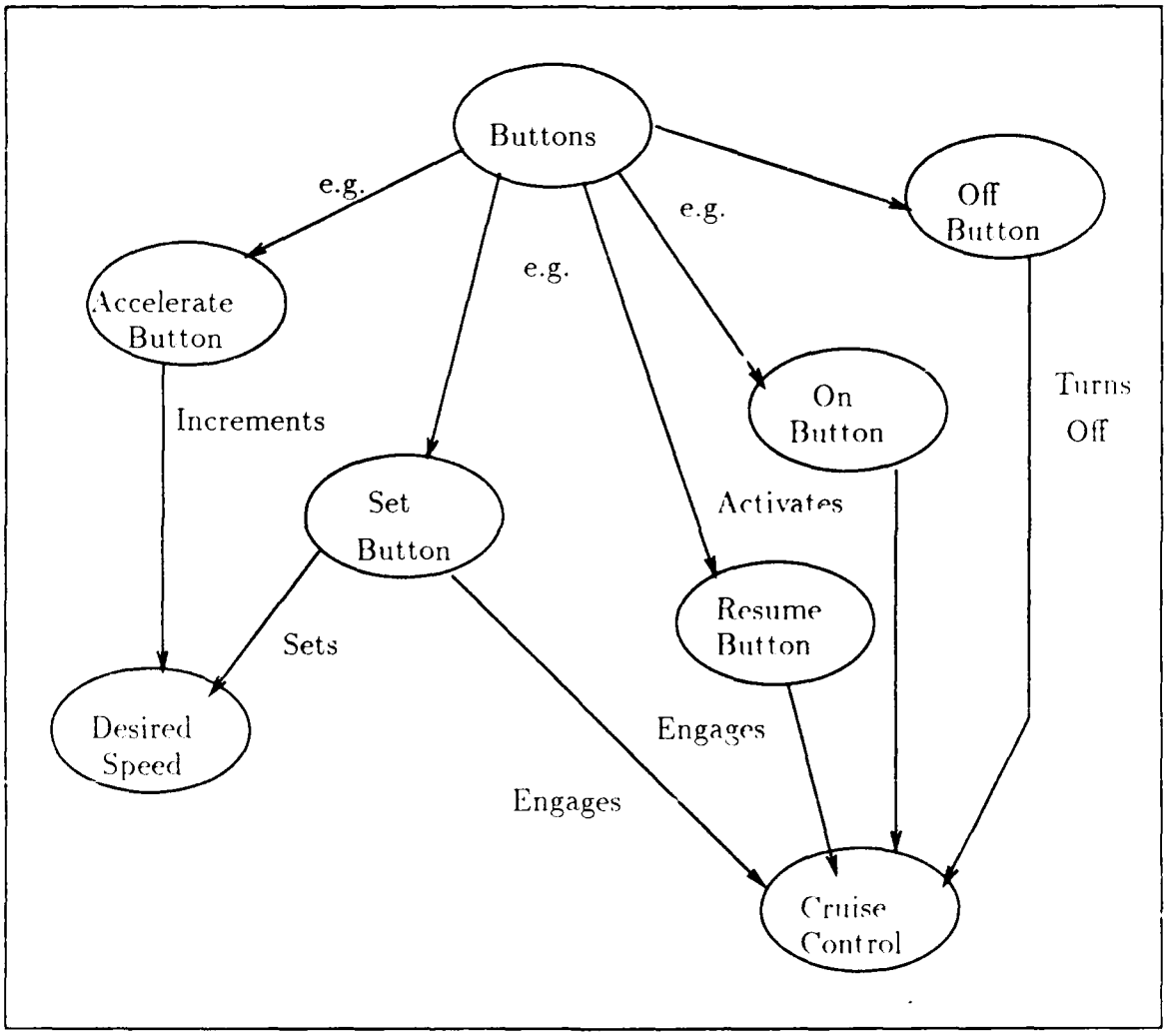
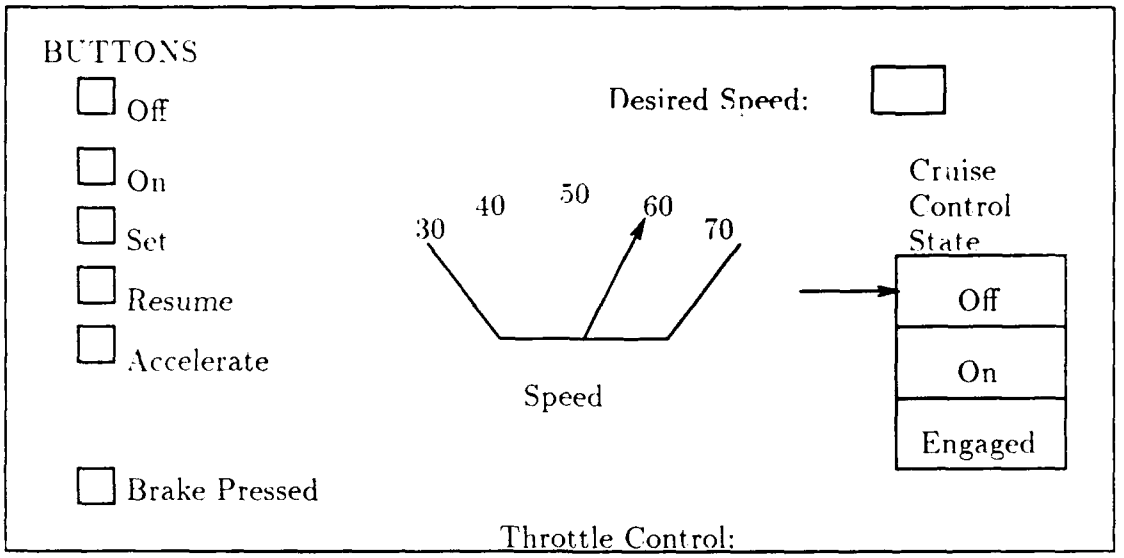


Figure 3.3. Concept Map: Cruise Control Buttons

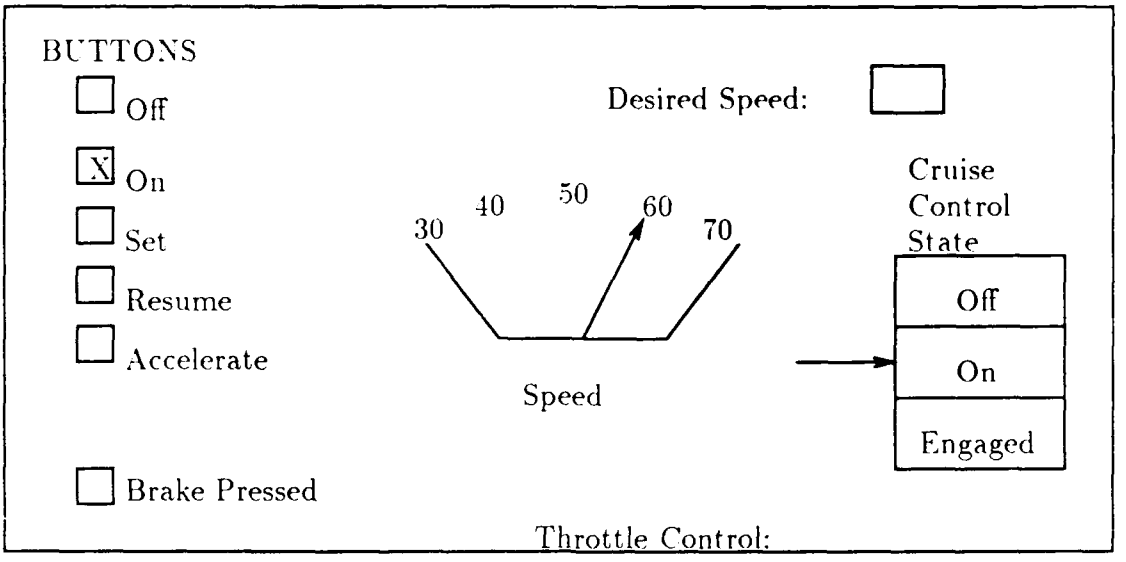


The cruise control powers up in the "off" state. Before it is turned on and set to a specific speed, the cruise control will remain idle. Only the "on" button will have any effect in this state.

Figure 3.4. Cruise Control Story Board: Initial Setting

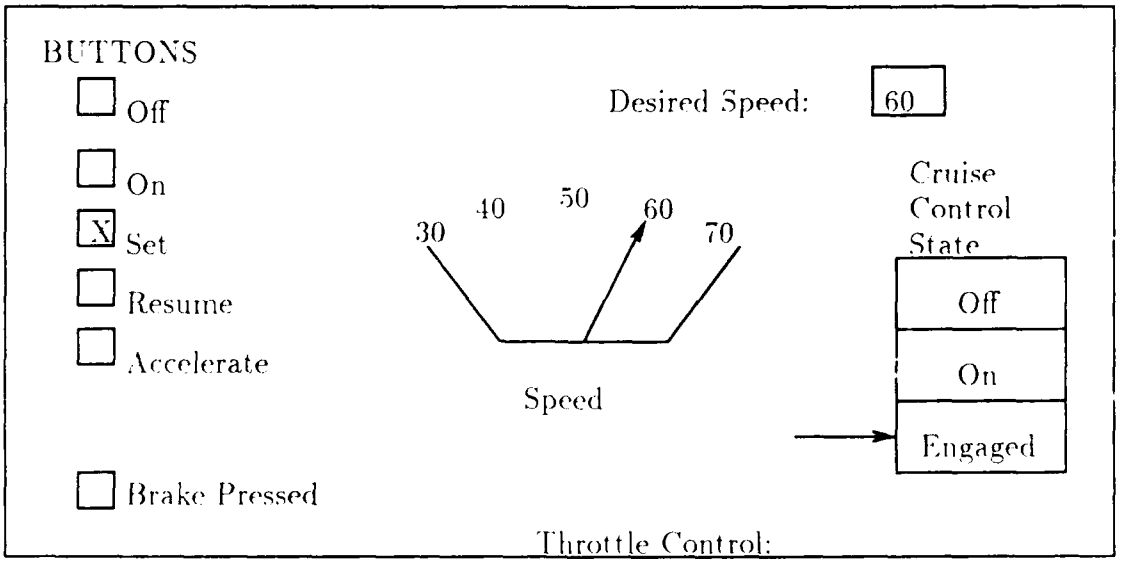
Step 1c: Outline the User Interface and Operational Scenarios with Story Boards.

There are no screen displays in the cruise control system. The speedometer does give an indication that the cruise control is operating, but the cruise control system does not directly manipulate it. However, story boards are useful for revealing the reaction of the cruise control system to various inputs from the environment. A few such scenarios are portrayed in the story boards in figures 3.4, 3.5, 3.6, 3.7, and 3.8. Other story boards could be added for a more complete description of the cruise control system.



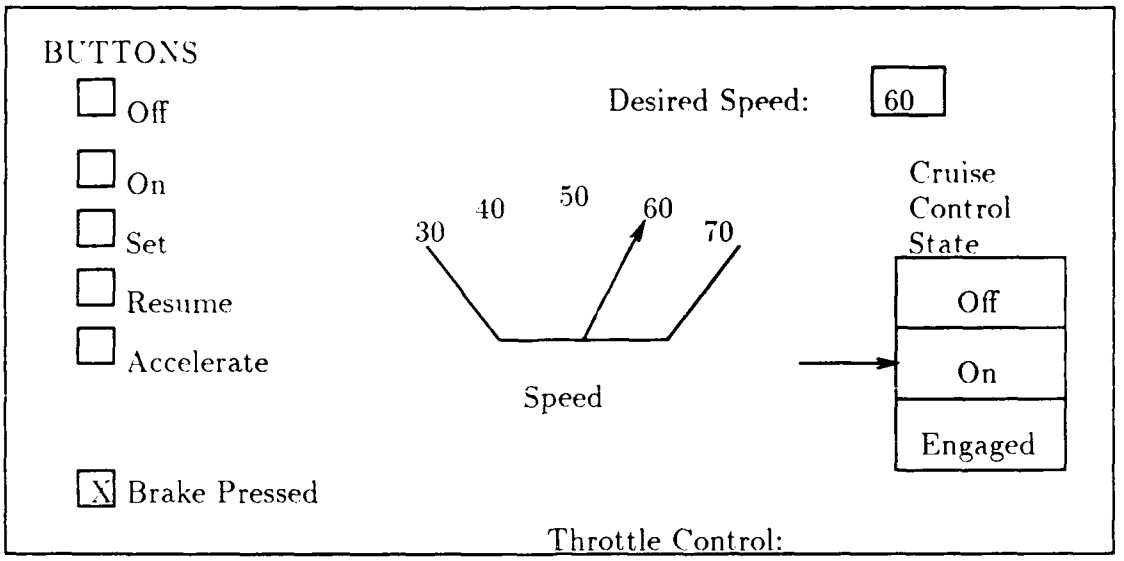
When the "on" button is pressed, the cruise control changes state, from "off" to "on". The cruise control system will now respond to the "set" button.

Figure 3.5. Cruise Control Story Board: On Button Pressed



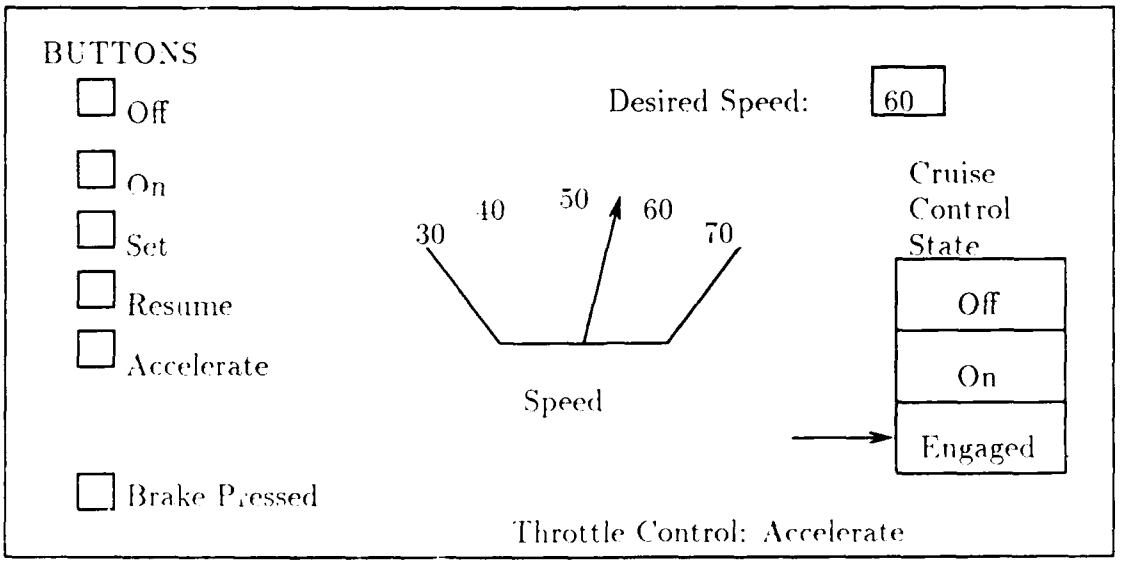
When the "set" button is pressed while the cruise control is in the "on" state, the cruise control enters the "engaged" state. The desired speed is set to the current speed, and the cruise control begins to control the throttle control when changes in speed are necessary.

Figure 3.6. Cruise Control Story Board: Set Button Pressed



If the brake is pressed while the cruise control is engaged, the system is disengaged and transitions to the "on" state. The throttle control is no longer active. The desired speed, however, remains set to its value in anticipation of a later command to resume the cruise control system.

Figure 3.7. Cruise Control Story Board: Brake Pressed



If the current speed drops below the desired speed while the cruise control is engaged, the cruise control system sends a signal to the throttle control to accelerate the vehicle. This accelerate signal will continue until the current speed is equal to or greater than the desired speed.

Figure 3.8. Cruise Control Story Board: Speed Drops

Step 1d: Produce an Event/Response List for the Software.

Event1: The on button is pressed.

Resp.1: The cruise control system is activated.

Maximum response time: 0.5 seconds.

E2: Set speed button is pressed.

R2a: Cruise control system is engaged.

R2b: Set the desired speed equal to the current speed.

Maximum response time: 0.25 seconds.

E3: Time to update the throttle position (periodic).

R3: If engaged, then set the throttle based on the current speed vs. the desired speed.

Projected event rate: 10 / second.

E4: Brake is pressed.

R4: Cruise control system is disengaged.

Maximum response time: 0.1 seconds.

E5: Resume button is pressed.

R5: Cruise control is engaged.

Maximum response time: 0.25 seconds.

E6: Accelerate button is pushed.

R6: Increment desired speed.

Maximum response time: 0.25 seconds.

E7: The off button is pressed.

R7a: Throttle control is disengaged.

R7b: Cruise control is deactivated.

Maximum response time: 0.1 seconds.

Step 1e: Identify Known Restrictions on the Software.

- The cruise control system object code must fit within 16K of memory.
- The cruise control system must disengage if the break is pressed at least 99.99999% of the time.

Step 1f: Identify any "metarequirements".

The maximum speed allowed for setting the cruise control system is 100 miles per hour.

3.3.3.2 Step Two: Model the Software Requirements in a Top-Down, Hierarchical Manner. Phase Two of the method consists of the following steps:

Step 2a: Draw an External Interface Diagram for the Software Component.

The external interface diagram is shown in figure 3.9.

Step 2b: Identify any High Level Actor Objects which Perform Some Overall Algorithm.

None. The algorithm of the cruise control system object is not complex enough to decompose. The object is documented in the object encyclopedia.

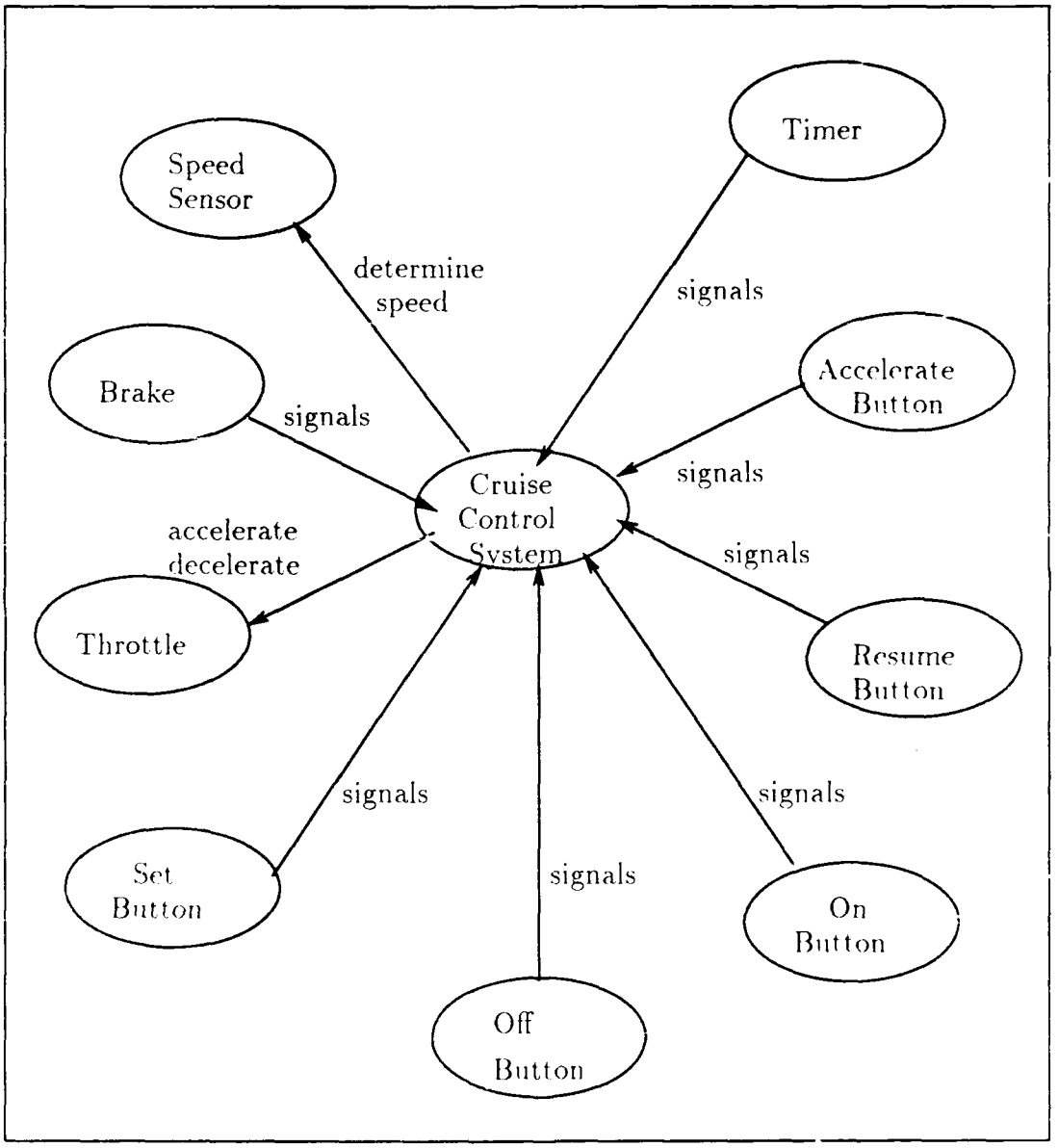


Figure 3.9. Cruise Control System: External Interface Diagram

Step 2c: Construct a Preliminary Object List.

```
Cruise Control

Throttle control

Speed
    Current Speed
    Desired Speed

Button
    Set Button
    On Button
    Off Button
    Resume Button
    Accelerate Button

Timer
```

Step 2d: Identify the Senders and Receivers of the Messages/Events.

Event1: The on button is pressed.
Sender: On Button
Receiver: Cruise Control

E2: Set speed button is pressed.
Sender: Set Button
Receiver: Cruise Control

R2a: Cruise control system is engaged. (Performed by Cruise Control)

R2b: Set the desired speed equal to the current speed. (Performed by Cruise Control)

E3: Time to update the throttle position (periodic).
Sender: Timer
Receiver: Cruise Control

E4: Brake is pressed.
Sender: Brake
Receiver: Cruise Control

E5: Resume button is pressed.
Sender: Resume Button
Receiver: Cruise Control

E6: Accelerate button is pushed.
Sender: Accelerate Button
Receiver: Cruise Control

E7: The off button is pressed.
Sender: Off Button
Receiver: Cruise Control

R7a: Throttle control is disengaged. (Performed by Cruise Control)

R7b: Cruise control is deactivated. (Performed by Cruise Control)

Step 2e: Document the Object Classes.

The following pages document representative classes of objects present in the cruise control problem.

Cruise Control Object

Textual Description:

The cruise control is the "brain" of the cruise control system. It keeps track of the state of the cruise control system and periodically updates the position of the throttle to maintain a constant vehicle speed.

Structure Diagram, Interface Diagram, and State Transition Diagram:

See figure 3.10, figure 3.11, and figure 3.12.

Description of messages received:

Break pressed	Signal that the break pedal has been pressed
Set button pushed	Signal that the set button has been pressed
Off button pushed	Signal that the Off button has been pressed
On button pushed	Signal that the On button has been pressed
Resume button pushed	Signal that the Resume button has been pressed
Accelerate button pushed	Signal that the Accelerate button has been pressed
Update Throttle	Signal that it is time to update the throttle position

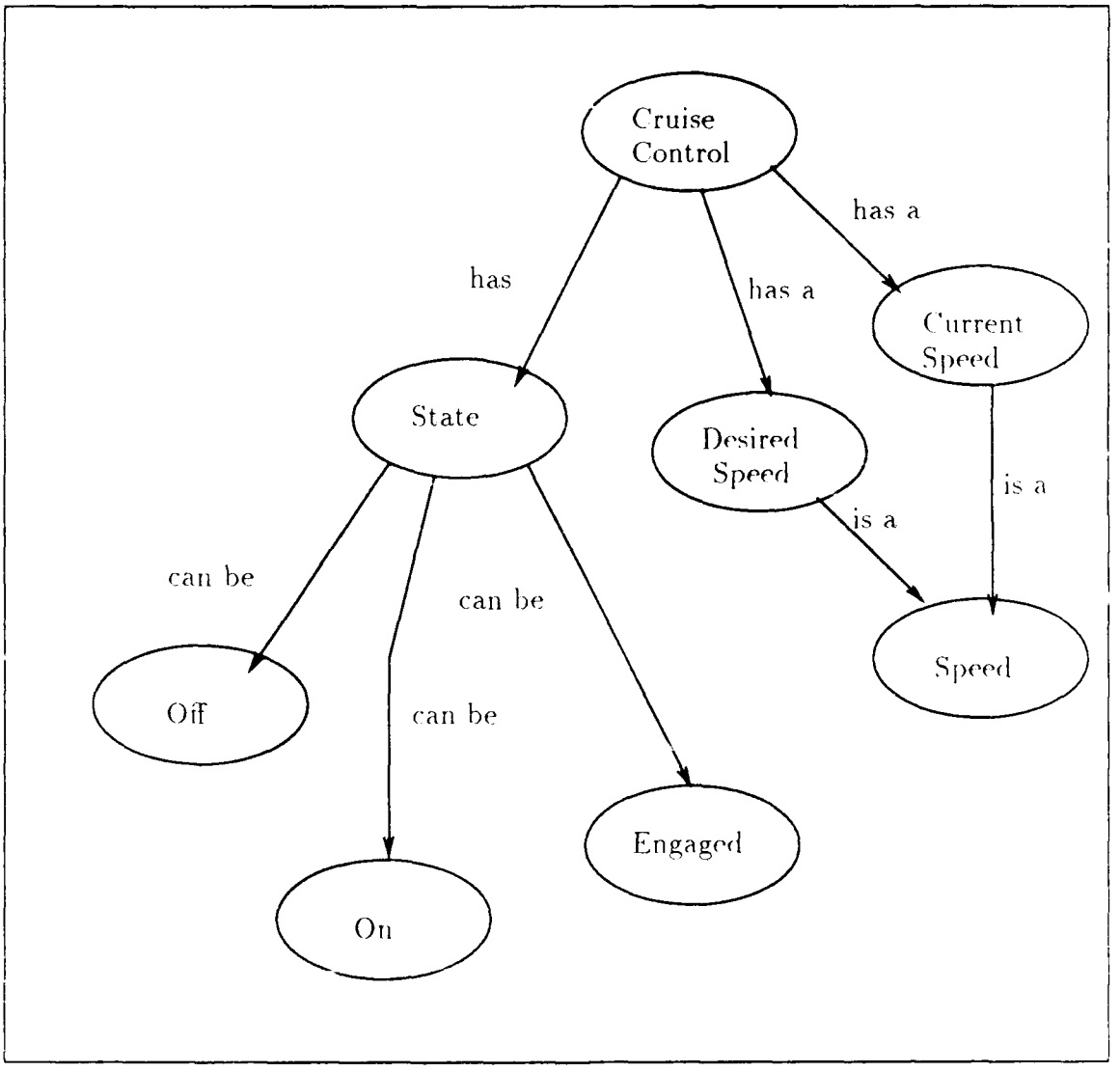


Figure 3.10. Cruise Control: Structure Diagram

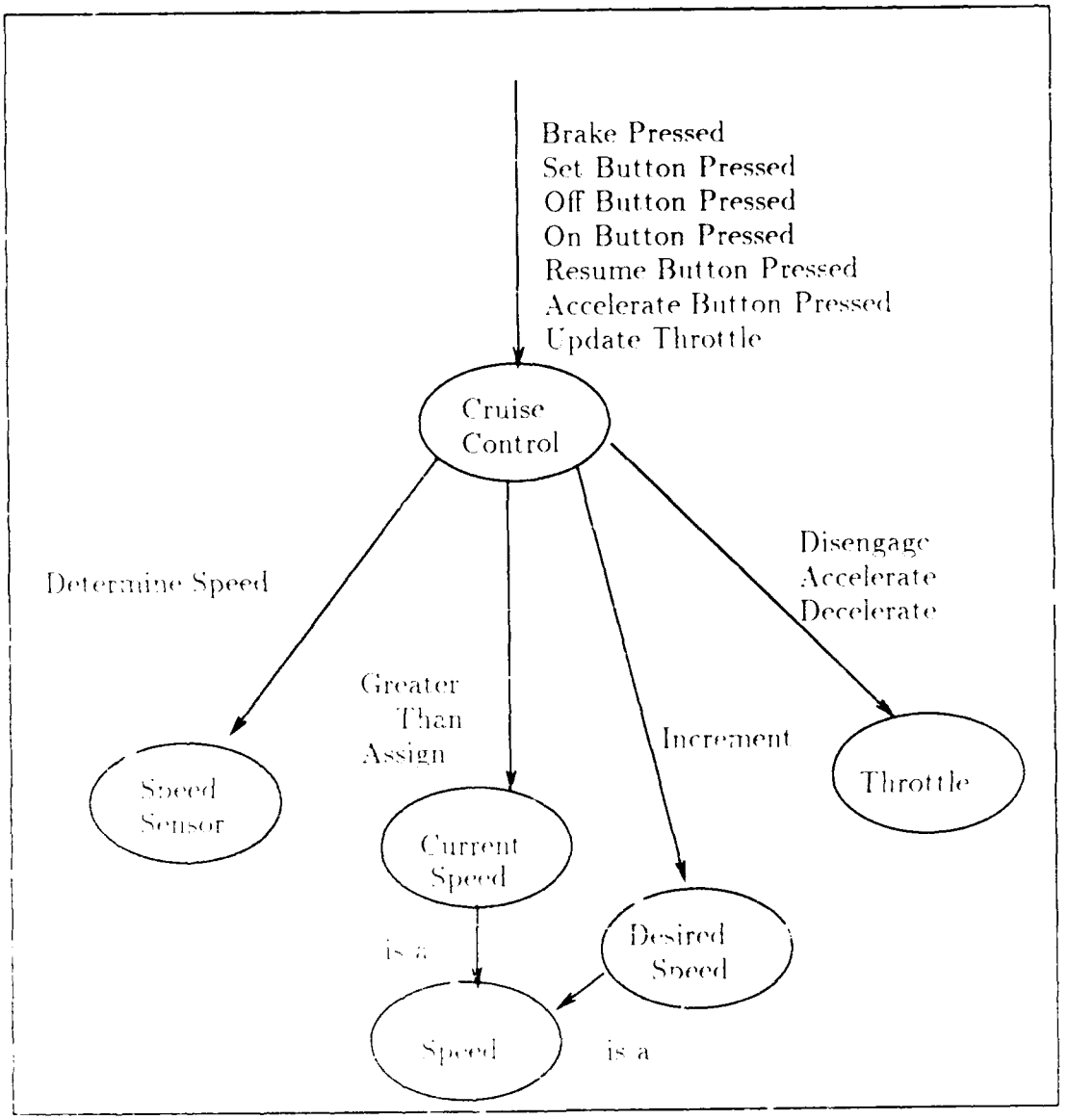


Figure 3.11. Cruise Control: Interface Diagram

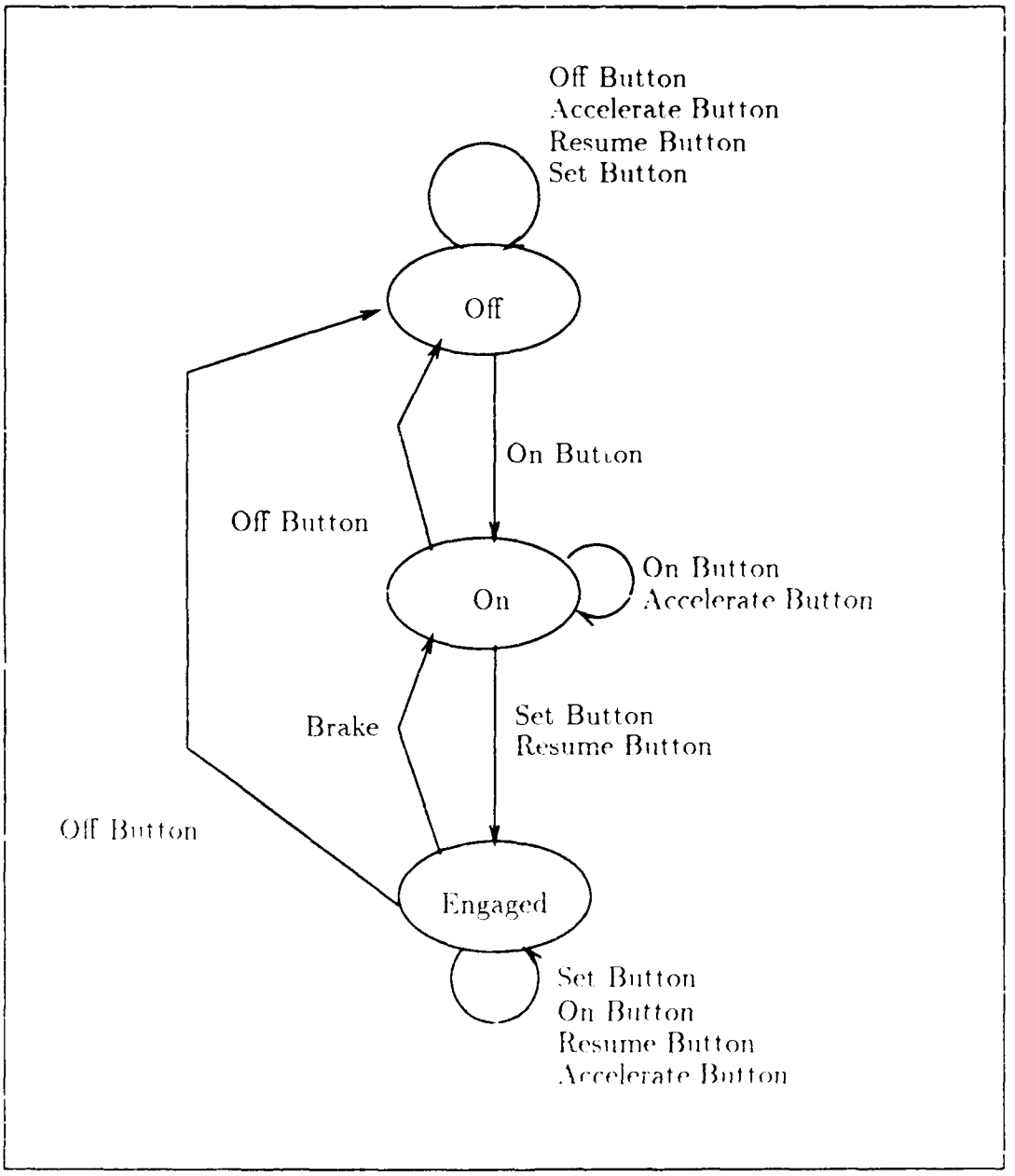


Figure 3.12. Cruise Control: State Transition Diagram

Description of messages sent:

Speed Sensor.Determine Speed	Get the current speed from the speed sensor
Speed.Greater Than	Determine if one speed is greater than another
Speed.Assign	Assign one value of class speed to another
Desired Speed.Increment	Increment the value of the desired vehicle speed
Throttle.Accelerate	Set the throttle to make the vehicle accelerate
Throttle.Decelerate	Set the throttle to make the vehicle decelerate
Throttle.Disengage	Release control of the vehicle throttle

Description of any state limitations:

The cruise control object must initialize in the "Off" state.

List of exported exceptions:

None.

List of exported constants:

None.

Reuse considerations:

This object is application specific.

Button Class

Textual Description:

The button class models a physical button. When the button is pushed, a signal is sent to some receiver. In this application, the receiver for all button objects is the cruise control object.

Structure Diagram, Interface Diagram, and State Transition Diagram:

See figures 3.13 and 3.14.

Description of messages received:

None.

Description of messages sent:

Signal Signal the receiver that the button has been pressed.

Description of any state limitations:

None.

List of exported exceptions:

None.

List of exported constants:

None.

List of objects in the class:

- On Button
- Off Button
- Set Button
- Accelerate Button
- Resume Button

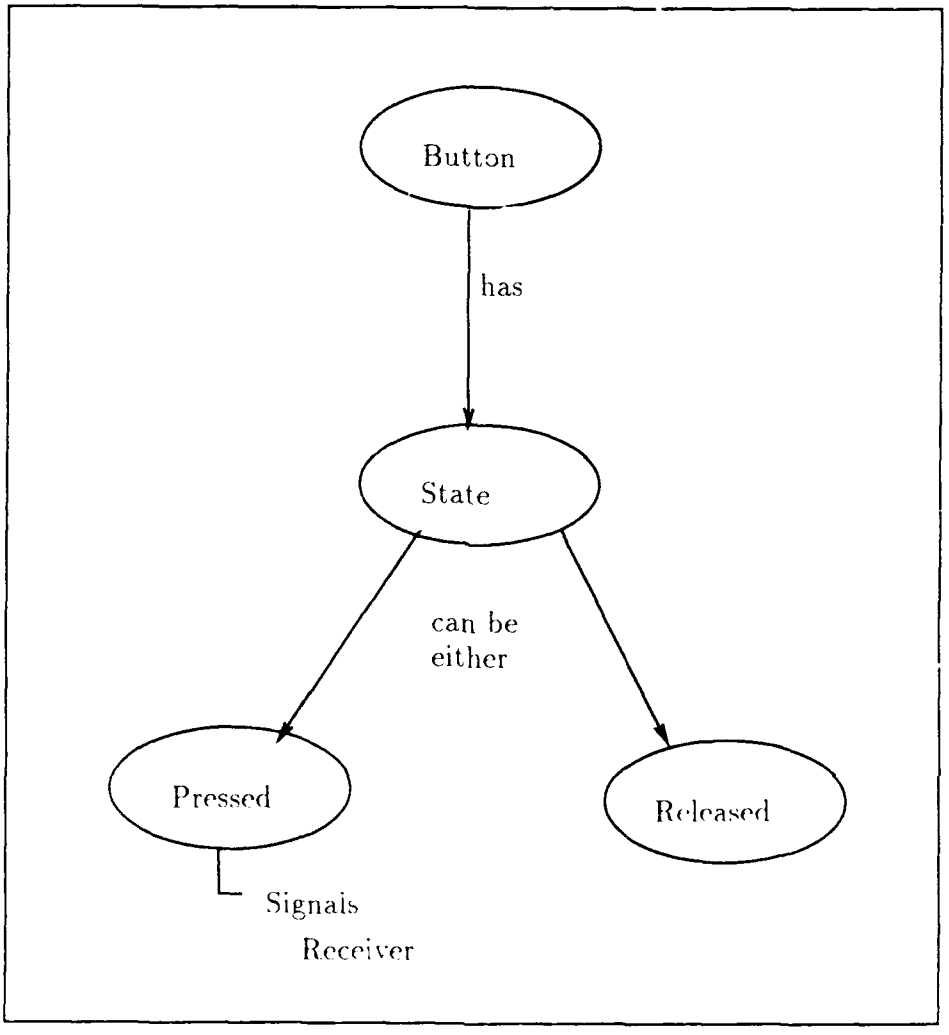


Figure 3.13. Button: Structure Diagram

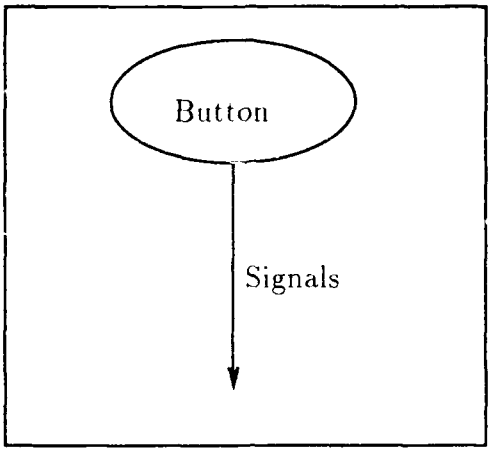


Figure 3.14. Button: Interface Diagram

Note: In each case the receiver of the signal is the cruise control object.

Reuse considerations:

This object is potentially reusable.

Speed Class

Textual Description:

This class describes objects which represent the speed of the vehicle. This class is based on the integer class.

Structure Diagram and Interface Diagram:

See figures 3.15 and 3.16.

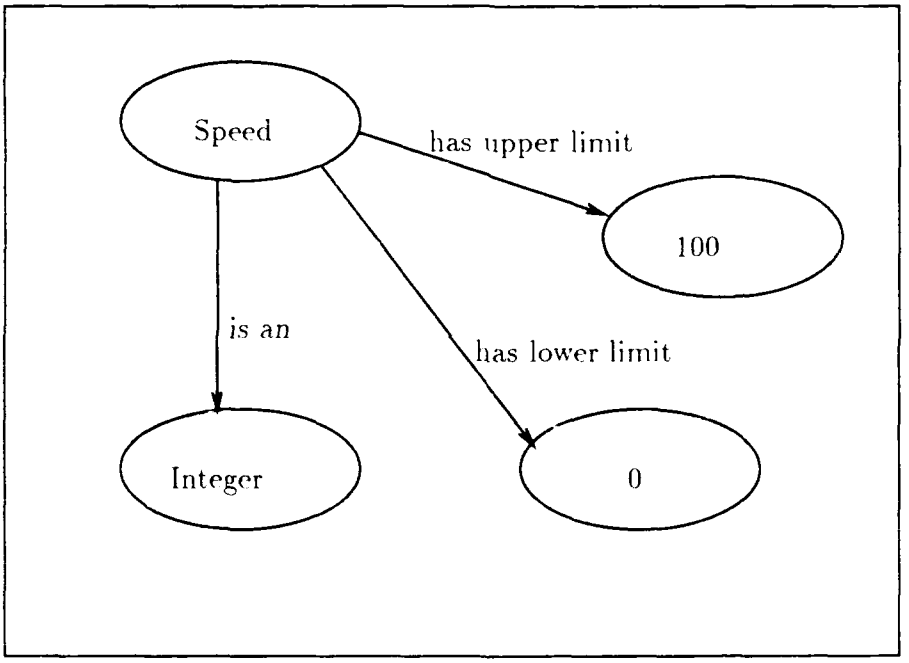


Figure 3.15. Speed: Structure Diagram

State Transition Diagram:

See figure 3.17.

Description of messages received:

Greater Than	Test if one speed is greater than another.
Increment	Increment the value of the speed.
Assignment	Assign one value of speed to another.

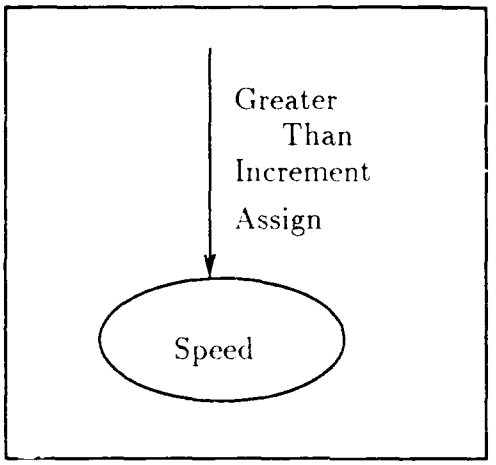


Figure 3.16. Speed: Interface Diagram

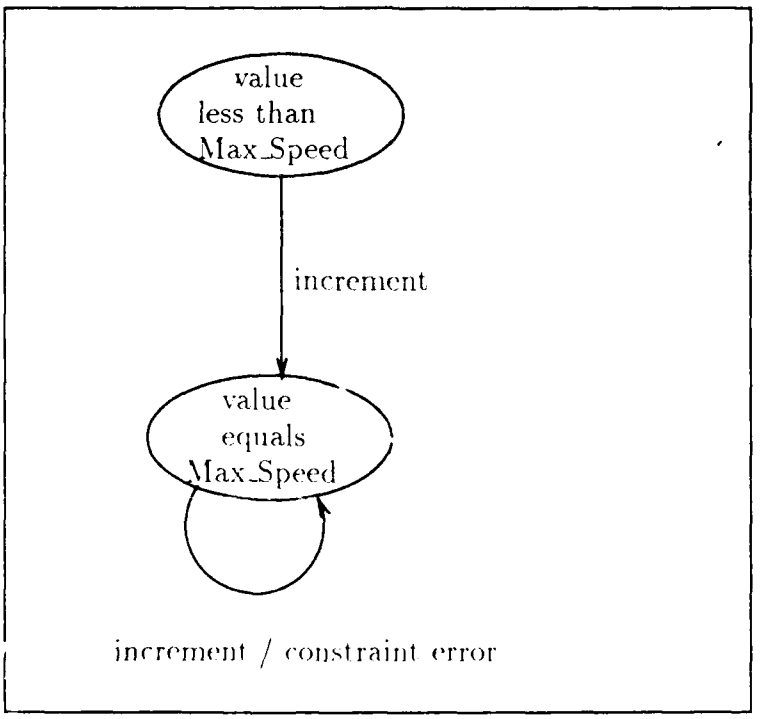


Figure 3.17. Speed: State Transition Diagram

Description of messages sent:

None.

Description of any state limitations:

An object of this class may have a value in the range 0..100.

List of exported exceptions:

Constraint Error	An attempt was made to increment a speed object which was at its maximum value

List of exported constants:

Maximum speed.

List of objects in the class:

- Current Speed
- Desired Speed

Reuse considerations:

This object has limited reuse potential.

Entries would also be placed in the object encyclopedia for the remainder of the objects in the object list.

3.4 *Mapping to an Object-Oriented Design*

The model produced by the OOA method maps directly into a Booch-flavored object-oriented design. Any high-level algorithm is documented, as are all classes of objects in the problem space. With the possible exception of certain external entities, each class of objects documented in the object encyclopedia will likely be part of the design. The class's entry in the object encyclopedia contains the information required to design the object.

Some details of the object classes were intentionally ignored in the analysis phase. For example, while OOA method identified the messages passed between objects, it made no attempt to define the arguments of these calls. The arguments of messages are often at a level of abstraction lower than the class being documented. Therefore, cataloging these arguments was deferred until the design phase.

Likewise, the nature of the implementation of each class of objects was not addressed during analysis. Whether objects are concurrent with other objects is one such implementation detail deferred until design. All objects are potentially concurrent—only efficiency considerations prevent the designer from actually implementing the objects in this manner. It is the designer's responsibility to determine the implementation of each object.

As the designer defines the software solution, he will undoubtedly discover more objects and operations to be implemented. For example, if the designer makes an object concurrent, he may have to add operations to initialize and terminate the object. The details of the solution will also present more object classes than were documented during analysis. In practice, information must be collected about these new object classes as well. Therefore, the designer may continue where the analyst left off, and use similar tools to those listed above in documenting objects identified during the design phase.

The tools and procedures defined in this chapter make up the object-oriented analysis (OOA) method. The domain expert and analyst may use paper and pencil to document this information. However, a computer-aided tool will greatly assist the analyst in creating and reviewing the documentation. Such a tool would also give more of a hierarchical nature to the entries in the object encyclopedia. The requirements for such a tool are outlined in chapter IV. The OOA method is more fully evaluated in chapter V, when the method is applied to a more comprehensive requirements analysis problem.

IV. Requirements for an Object-Oriented Analysis Tool

The previous chapter suggested the creation of a software tool to manage the products and process of the object-oriented analysis method. This chapter describes a user interface and set of guidelines for the design of such a tool. The purpose of this description is to show the potential benefits the OOA method can receive from automated support. Although some of the tools from the OOA method are used in this description, this chapter is *not* an example of the OOA method—a complete application on an example problem is presented in chapter V. A more complete specification and design of the OOA tool is recommended as a future project (see section 6.3).

4.1 Framework for OOA Tool Description

The Object-Oriented Analysis Tool contains many characteristics of a *Decision Support System (DSS)*. The tool is aimed at providing support to the analyst in his attempt to model the software requirements. The proposed OOA tool, however, does not seem to fit the definition of a "true" DSS. A typical DSS contains some type of model that draws upon a data base of information in order to provide a *quantitative assessment* which will assist a decision maker. At the beginning of the software requirements analysis activity, there is little information that would be in an existing database (other than a library of reusable components) and no formal model to draw upon the information that does exist. The OOA method is more of a *creative* process than an analytic one, guided by heuristics instead of models. Nevertheless, a DSS framework can be used to identify the nature of the OOA tool's user interface.

Sprague and Carlson have proposed a process-independent approach for identifying the capabilities of a DSS. This approach emphasizes the consideration of Representations, Operations, Memory aids, and Control mechanisms (ROMC). *Rep-*

resentations communicate information about the problem to the user, often using a report or graphical format. The DSS should provide a set of *operations* to manipulate the information in the representations. *Memory aids* assist or guide the user in applying the operations on the representations. The *control mechanisms* enable the user to direct the session with the tool, and get from one set of representations to another [Sprague and Carlson, 1982:96]. The ROMC principles guided the description of the OOA tool user interface presented here.

The chapter uses two previously describe tools, the concept map and storyboard, to present the OOA tool description. The concept maps identify the elements of the object-oriented requirements specification produced by the OOA method. The maps also identify the information (memory aids) upon which a specific element of the specification depends. The storyboards describe a proposed user interface for the OOA tool. They were created to describe the automated support for developing each of the elements illustrated in the concept maps. The development of each storyboard considered the representations, operations, memory aids, and control mechanisms appropriate for that step in the OOA method.

4.2 Relationships Among Models in the Object-Oriented Analysis Method

Many of the steps in the Object-Oriented Analysis (OOA) method are based on the work of previous steps. With this in mind, one can draw a set of concept maps reflecting these relationships among the models developed in the method.

Figure 4.2 shows a concept map describing the overall OOA method. As defined in the previous chapter, the OOA method has two major steps: capturing the software requirements from the domain expert, and structuring those requirements into a form suitable for design. The later process of object-oriented design is based primarily on this structured representation, though the initial concept maps and storyboards may also influence the designer.

The elements of the initial, unstructured model of the requirements are illus-

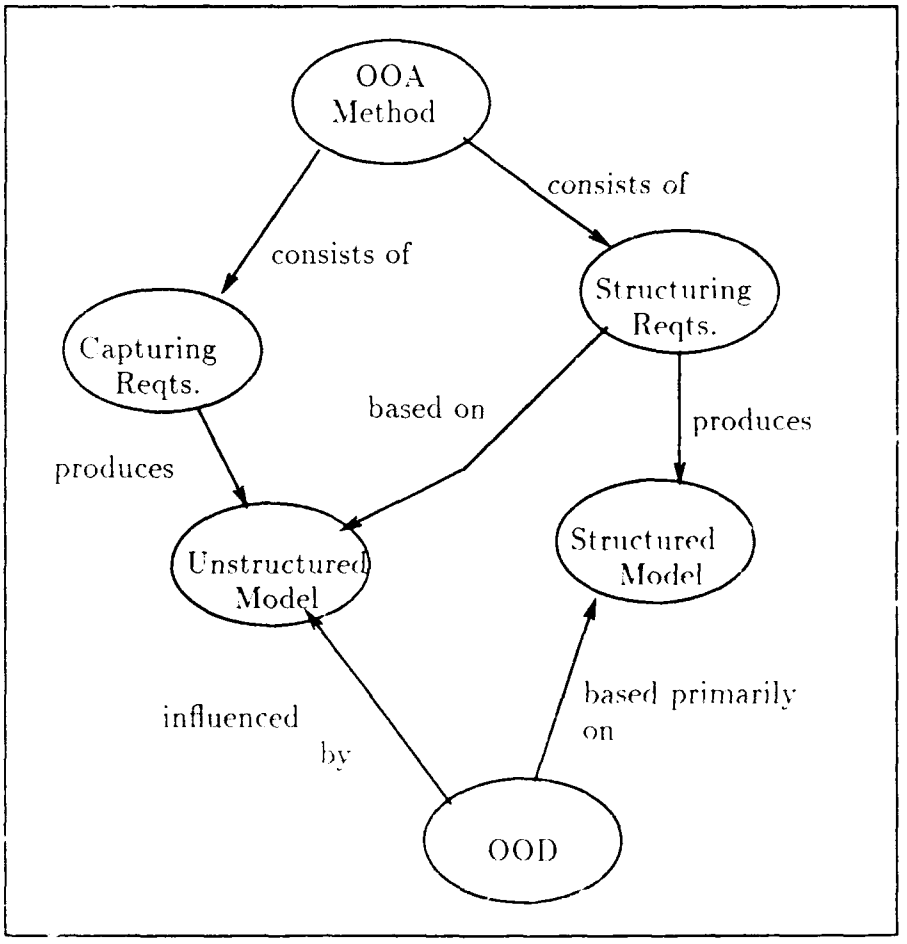


Figure 4.1. Concept Map: OOA Method

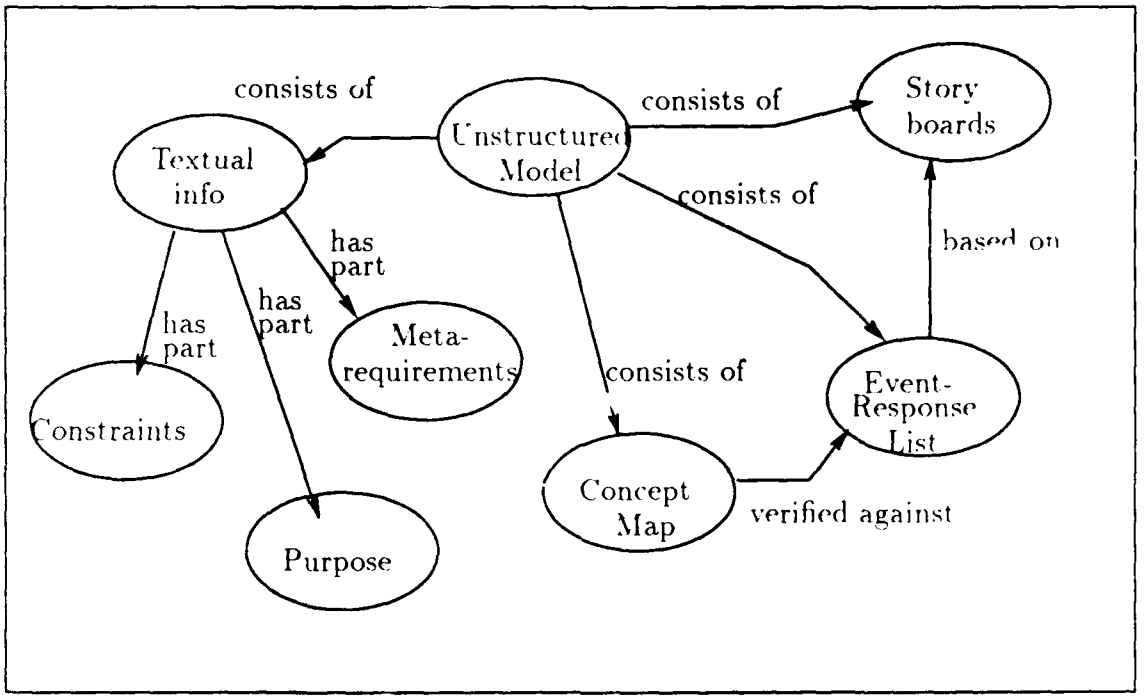


Figure 4.2. Concept Map: Capturing the Requirements

trated in figures 4.2, 4.3, and 4.4. These maps point out some of the relationships among the different models. For example, the event/response list is based on the domain expert's concept map and storyboards, while the concept map is verified against the event/response list. An OOA tool can assist the analyst in applying the method steps by informing the analyst of these relationships among models, and providing a means of displaying previous information upon which a specific model is based.

The models developed in the second step of the OOA method display a similar reliance upon previous models. For example, the external interface diagram is based on both the domain expert's concept maps and the event/response list. Figures 4.5, and 4.6 describe the primary relationships between these models in further detail.

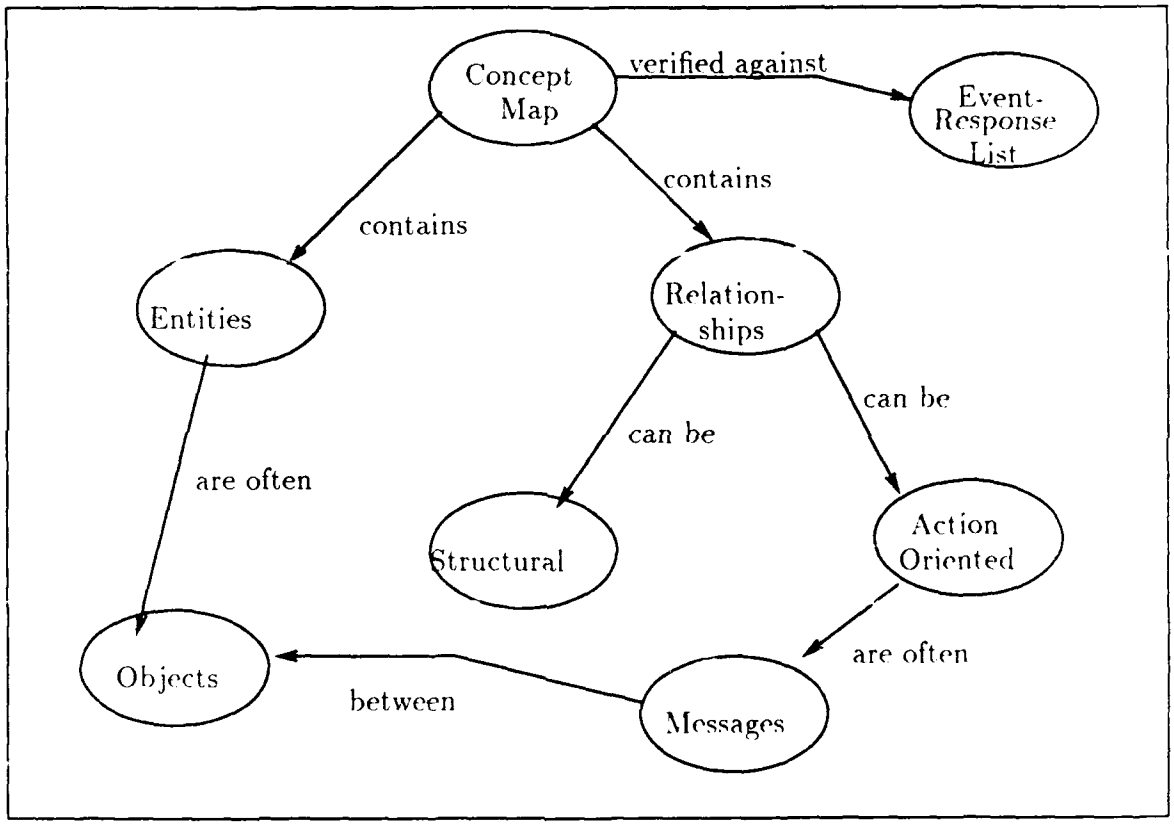


Figure 4.3. Concept Map: The Unstructured Concept Map

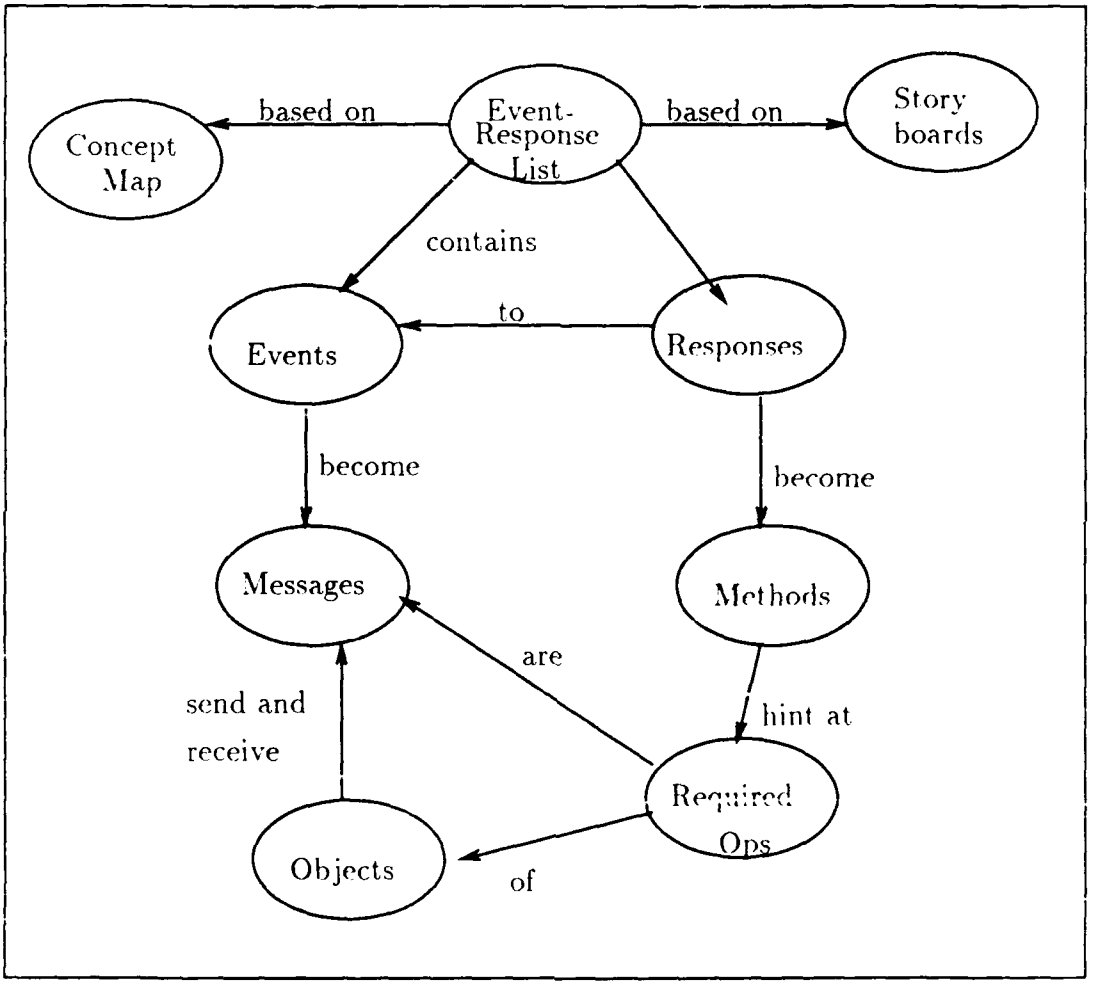


Figure 4.4. Concept Map: The Event/Response List

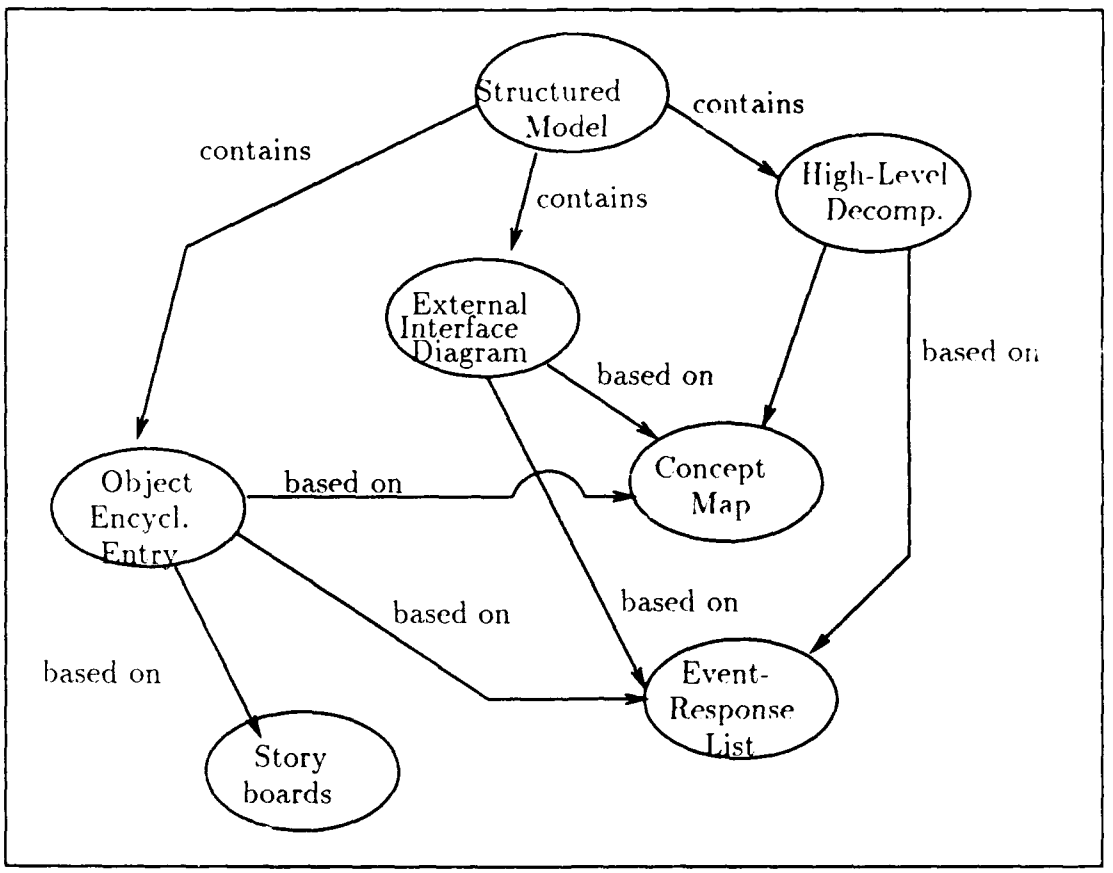


Figure 4.5. Concept Map: Structuring the Requirements

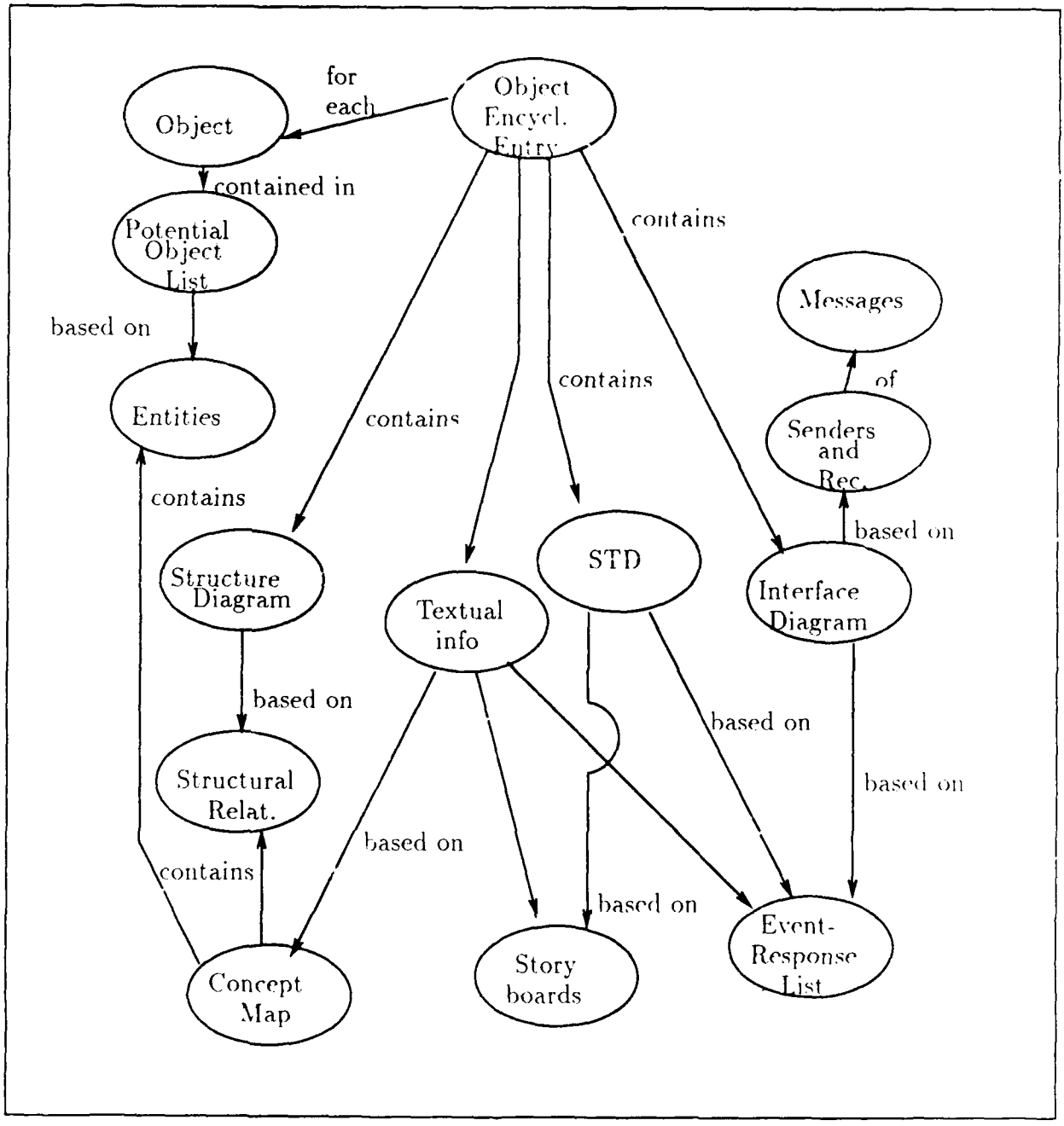


Figure 4.6. Concept Map: The Object Encyclopedia

4.3 *General Requirements for an Object-Oriented Analysis Tool*

First and foremost, the OOA tool should be "user friendly". This is especially important in light of the fact that the domain expert may be the one using the tool to enter initial concept maps, storyboards, and event/response lists. Therefore, the OOA tool should run in a graphical windowing environment, with user input allowed through menus and a mouse. Use of an existing windowing environment also provides the user the ability to customize the size and shape of the tools windows. This flexibility enhances the tool's ability to support a range of user preferences.

Because the models developed during the course of analysis depend on previous representations, the OOA tool should enable the user to view and edit multiple views simultaneously. Therefore, the windowing environment should enable the user to open multiple windows with different models of the requirements.

To assist the novice user, the system should provide a reference to context sensitive help windows. This help should be of two kinds. It should: 1) provide assistance on the use of the OOA tool itself, and 2) suggest guidelines for applying the steps of the OOA method. Also, the OOA tool should be consistent in the presentation of screens and menus. Menu selections that are present in all windows (e. g. Help) should be positioned in the same relative position in each menu. This positioning will minimize the "learning curve" of a new user.

Occasionally, the analyst may have a random thought about a topic not directly related to the model he is currently working on. Therefore, the OOA tool should have a notepad capability to provide an easy way to capture these thoughts. The notepad may also be used to record any difficulties encountered while using the OOA tool.

In cases where there is overlap of information among different models in the OOA method, the OOA tool should ensure that the different representations are consistent. For example, the OOA tool could ensure that each class of object in

the list of potential objects is documented with an entry in the object encyclopedia. Other potential cross checks the tool could perform include:

- verifying that each event in the event/response list has been cataloged with a message sender and receiver.
- ensuring that each message in the list of messages shows up on the interface diagram for some object class.
- confirming each message sent or received by a class of objects shows up on both the interface diagram and the textual description for that class.
- verifying that the list of messages received by a particular class includes each of the messages identified in other classes as being sent to that object.

The general requirements cataloged above (along with the concept maps presented earlier in this chapter) are sufficient to carve out a set of storyboards which illustrate the "look and feel" of an OOA tool.

4.4 Storyboards of the Object-Oriented Analysis Tool

The relationships between model elements shown in the concept maps and discussed in section 4.2 furnish insight into the requirements for a software tool to assist in applying the OOA method. Each step in the OOA method can be supported through a menu choice of the OOA tool. The tool will assist the analyst in each step by providing access to the information that each step is based on.

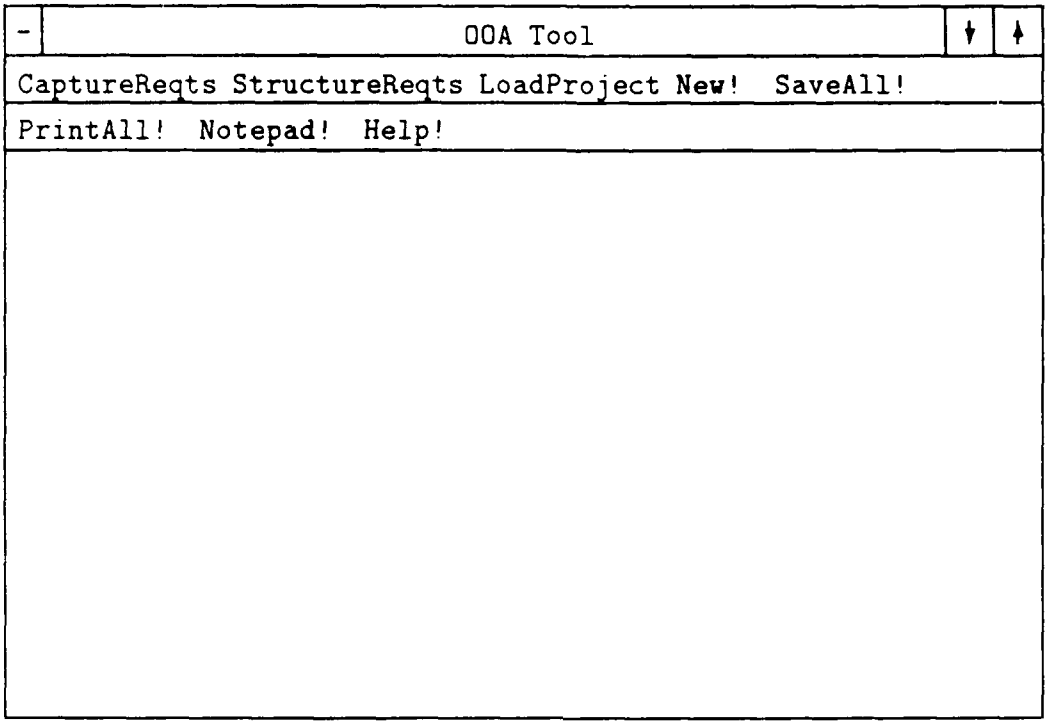
The initial screen of the OOA tool is shown in figure 4.7. This window has a number of characteristics that are common to other windows presented by the OOA tool. Windows of this nature can be constructed with a number of different windowing systems; the format shown is characteristic of Microsoft Windows for the IBM PC class computer. Clicking the mouse on the small box in the upper left corner of the window presents a menu of window commands, such as resizing,

moving, or closing the window. By closing the main window, the user concludes a session with the OOA tool. The arrowed boxes in the upper right hand corner allow the user to expand the window to cover the entire screen, or to shrink the window into an icon. Other menu choices are presented in rows across the top of the window. An exclamation point at the end of a menu choice denotes an action which takes place immediately when choice is selected. A menu choice lacking the trailing exclamation point will request more information from the user before the action is taken. Windows may also have scroll bars to move the window over a larger underlying drawing or text.

The initial screen of the OOA tool integrates all steps of the OOA method. From this display, the user has access to the models developed in both step one (Capture Requirements) and step two (Structure Requirements) of the OOA method. From this window, the user may also load an existing project, or clear the tool for a new project. Commands also allow saving the project, printing all models and documentation, and access to the notepad and help functions.

4.4.1 Capturing Software Requirements. Figure 4.8 displays the menu choices for tools which support capturing the software requirements under step one of the OOA method. If the user selects the *Text!* option, the window shown in figure 4.9 is created. This window, and similar windows described later, do not take up the entire screen. This allows the user access to the main menu, and thus other models developed during the analysis. Of course, the analyst may always click the mouse on the "up" arrow in the box in the upper right corner to expand the window to cover the entire screen.

From the *CaptureReqts* menu, the user may also develop and/or view a concept map (see figure 4.10), storyboard (figure 4.11), or event/response list (figure 4.12).



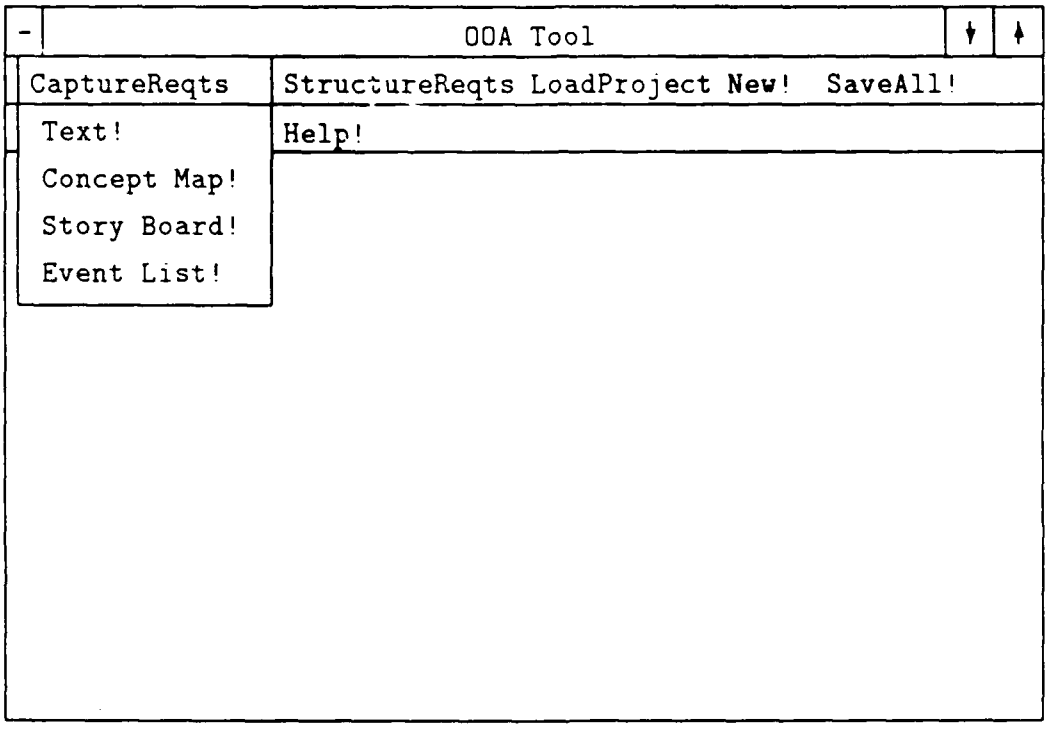
This is the main screen of the OOA tool. The window format and commands of this window are similar to those used for other screen displays. Clicking the left mouse button on the small box in the upper left corner of the window presents a menu of window commands, such as *close*, or *resize*. Clicking on the arrow boxes in the upper right hand of the window allows the user to maximize the window to cover the entire screen, and shrink the window into an icon.

The second and third lines of the window display menu choices that may be selected with the mouse. Any choice ending in an exclamation point initiates an immediate action, while those without the exclamation point present another menu of choices. All windows have menu options *Notepad!* and *Help!*. These options allow the user to access a notepad to record random thoughts while using the tool, and context sensitive help and guidelines about the system and OOA method.

The main menu contains the following options:

- *CaptureReqts*: Present the "Capture Requirements" menu (see figure 4.8) to perform actions in step one of the OOA method.
- *StructureReqts*: Present the "Structure Requirements" menu (see figure 4.13) to perform actions in step two of the OOA method.
- *LoadProject*: Present a list of project names that the user may load.
- *New!*: Clears the OOA tool for a new project.
- *SaveAll!*: Save all aspects of the project to disk. If no project was initially loaded, the user will be prompted for the project name.
- *PrintAll!*: Print all project documentation.

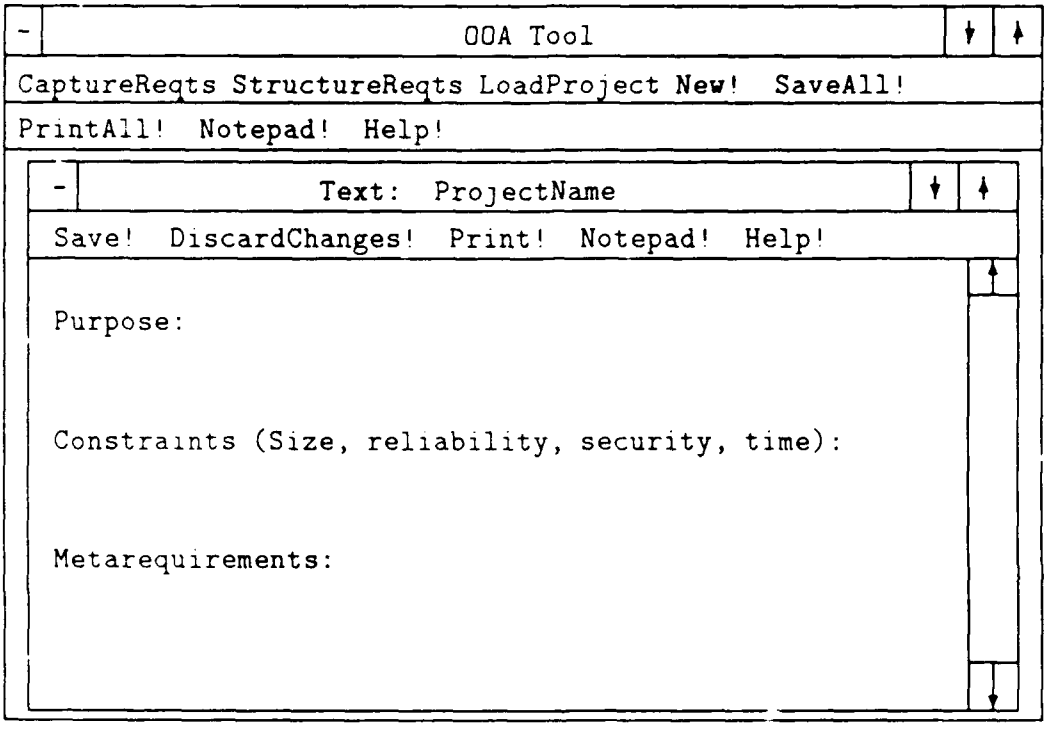
Figure 4.7. OOA Tool Storyboard: Main Tool



The "Capture Requirements" menu allows the user to make the following choices for step one of the OOA method:

- *Text!*: Document textual information about the system (see figure 4.9).
- *Concept Map!*: Draw a set of concept maps (see figure 4.10).
- Storyboard!: Draw a set of storyboards (see figure 4.11).
- Event List!: Create an event/response list (see figure 4.12).

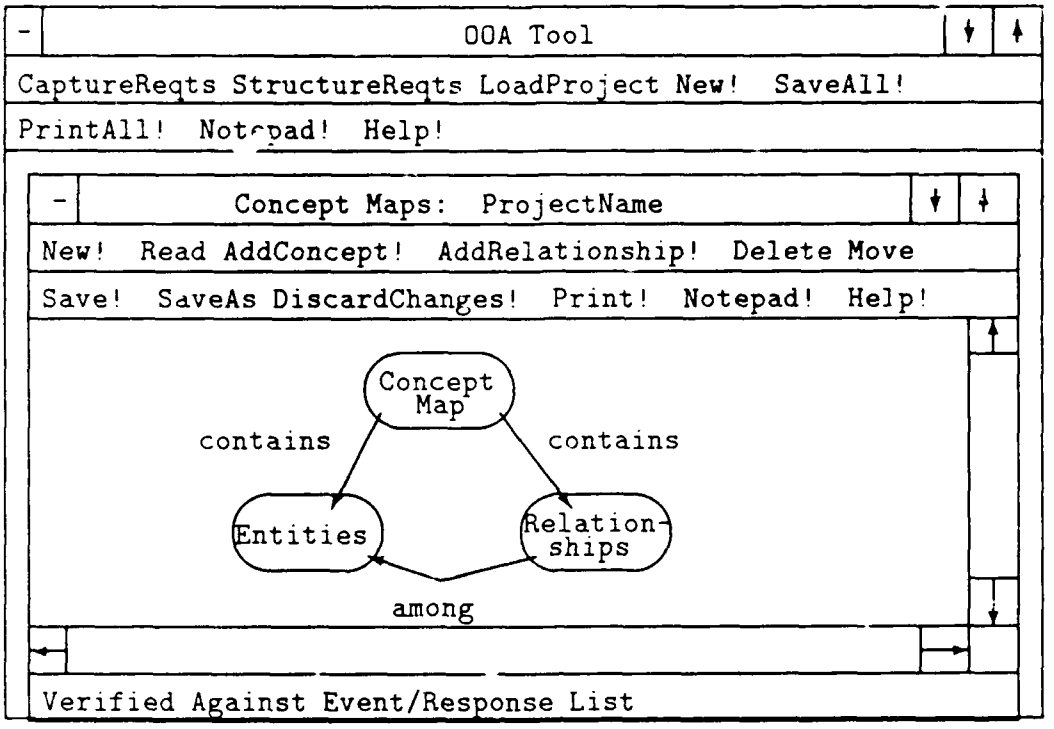
Figure 4.8. OOA Tool Storyboard: Capture Requirements Menu



When the *Text!* selection is chosen from the "Capture Requirements" menu, the textual information input window appears. This window contains a template of the textual information that should be captured from the domain expert. The initial position of the window allows the user to access the main menu to get other information about the project. The scroll bar on the right hand side of the window allows the user to scroll through the text with a click of the mouse button.
The following menu choices are specific to this window:

- *Save!*: Save the current textual information.
- *DiscardChanges!*: Discard changes to the textual information since the last save.
- *Print!*: Print the textual information.

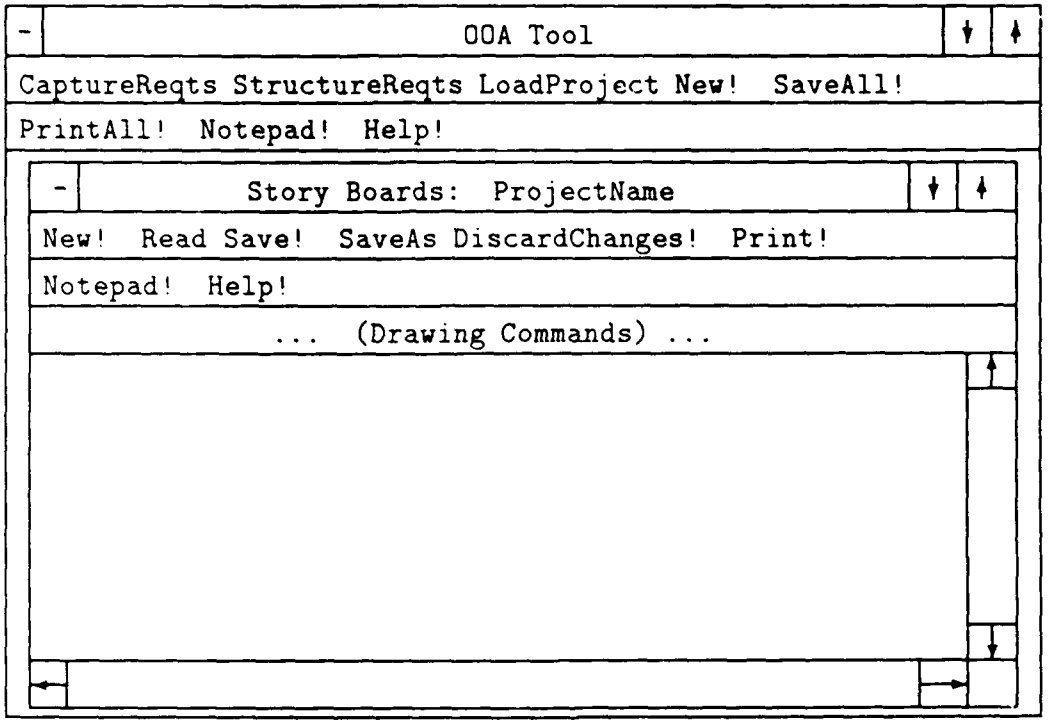
Figure 4.9. OOA Tool Storyboard: Textual Information



When the *Concept Map!* selection is chosen from the "Capture Requirements" menu, the concept map window appears. This window contains commands which allow the user to draw concept maps. The initial position of the window allows the user to access the main menu to get other information about the project. The bottom line of the window displays the other method steps upon which the concept maps are based. The scroll bar on the bottom and right hand sides of the window allow the user to scroll through the concept map with a click of the mouse button. The following menu choices are specific to this window:

- *New!*: Initializes the window to draw a new concept map.
- *Read*: Reads a concept map from a file.
- *AddConcept!*: Adds a concept to the map. The user is prompted for a label for the concept, and is allowed to position the concept with the mouse.
- *AddRelationship!*: Adds a relationship between two concepts. The user is prompted for the label to attach to the relationship, and is allowed to position the label with the mouse.
- *Delete*: This selection is used to delete a concept or relationship. After selecting this option, the user clicks the mouse on the concept or relationship to delete.
- *Move*: This selection is used to move a concept. After selecting this option, the user clicks on the concept to move, then drags it to its new location.
- *Save!*: Save the current concept map.
- *SaveAs!*: Save the concept map in a different file. The user is prompted for the name of the new file.
- *DiscardChanges!*: Discard changes to the concept map since the last save.
- *Print!*: Print the current concept map.

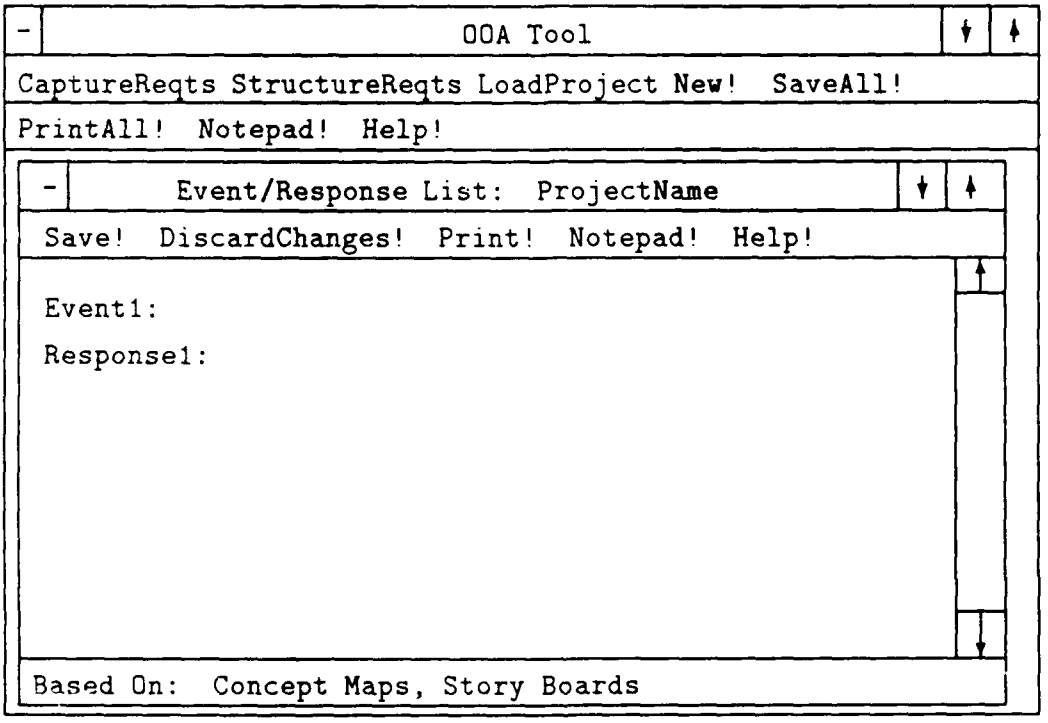
Figure 4.10. OOA Tool Storyboard: Concept Maps



When the *Storyboard!* selection is chosen from the "Capture Requirements" menu, the storyboard window appears. This window connects to a drawing package which allows the user to draw free-format graphics for storyboards. The initial position of the window allows the user to access the main menu to get other information about the project. The scroll bar on the bottom and right hand sides of the window allow the user to scroll through the concept map with a click of the mouse button.
The following menu choices are specific to this window:

- *New!*: Initializes the window to draw a storyboard frame.
- *Read!*: Reads a storyboard frame from a file.
- *Save!*: Save the current storyboard frame.
- *SaveAs!*: Save the storyboard frame in a different file. The user is prompted for the name of the new file.
- *DiscardChanges!*: Discard changes to the storyboard frame since the last save.
- *Print!*: Print the current storyboard frame.
- *Drawing Commands*: Drawing commands specific to an existing free-format drawing program will be included.

Figure 4.11. OOA Tool Storyboard: Storyboard Window

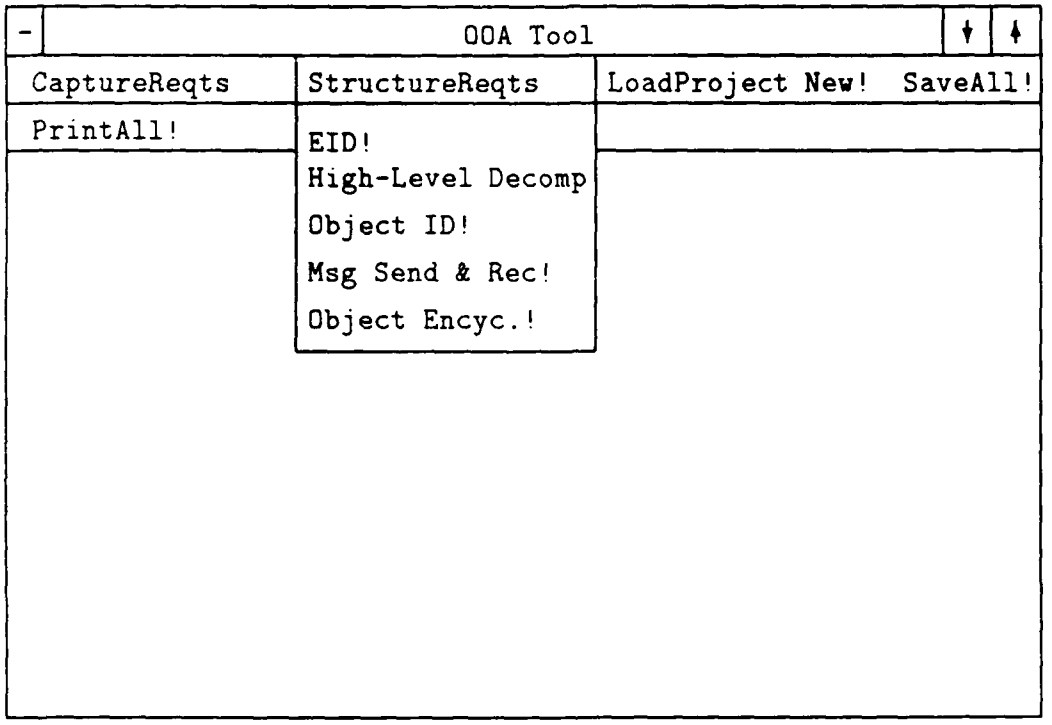


When the *Event List!* selection is chosen from the "Capture Requirements" menu, the Event/Response List window appears. This window contains templates for entering events to which the software must respond, and their corresponding responses. The initial position of the window allows the user to access the main menu to get other information about the project. The bottom line of the window displays the other method steps upon which the event/response list is based. The scroll bar on the right hand side of the window allows the user to scroll through the list with a click of the mouse button.
The following menu choices are specific to this window:

- *Save!*: Save the current concept map.
- *DiscardChanges!*: Discard changes to the concept map since the last save.
- *Print!*: Print the current concept map.

Figure 4.12. OOA Tool Storyboard: Event/Response List

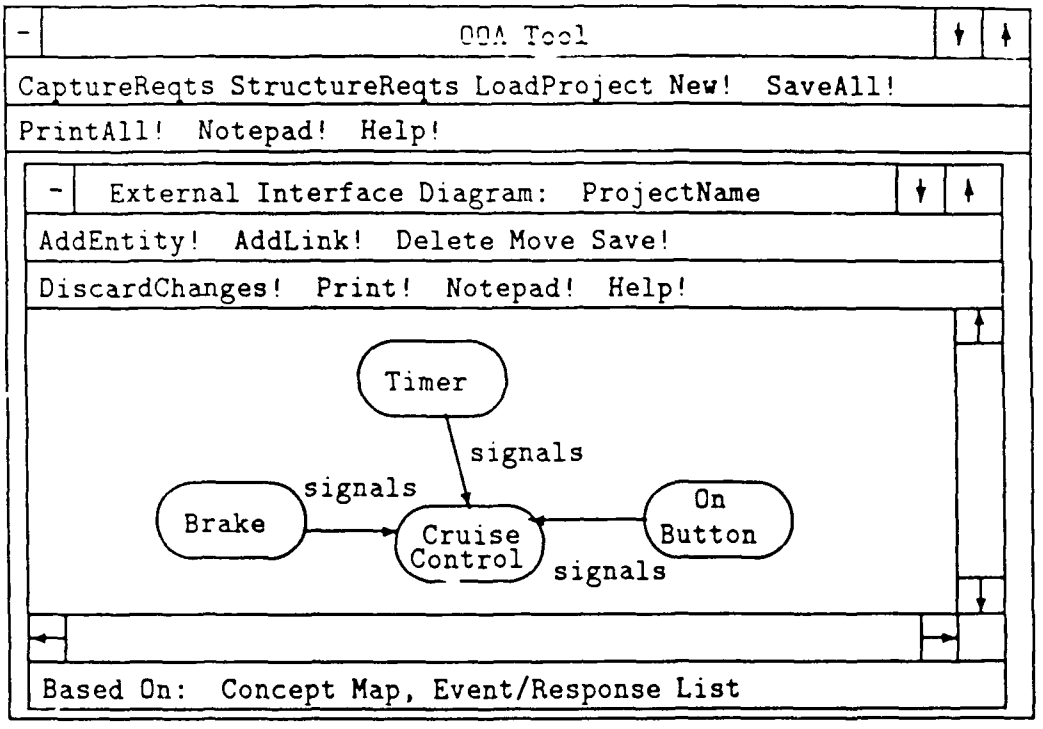
4.4.2 Structuring Software Requirements. The *StructureReqts* menu (see figure 4.13) provides a number of tools which can assist the analyst in adding structure to the requirements to make them sufficient for the object-oriented design phase. From this menu, the user may choose to draw or view an external interface diagram (see figure 4.14), decompose any high-level algorithm (see figure 4.15), list potential objects and classes (figure 4.16), list message senders and receivers (figure 4.17), or edit or view entries in the object encyclopedia (see figure 4.18). Each of these options is discussed further in the text corresponding to the storyboards in the figures.



The "Structure Requirements" menu allows the user to make the following choices for step two of the OOA method:

- *EID!*: Draw an external interface diagram (see figure 4.14).
- *High-Level Decomp!*: Draw a structure diagram depicting the decomposition of the high-level algorithm (see figure 4.15).
- *Object ID!*: Enter a list of potential objects in the solution of the system (see figure 4.16)
- *Msg Send & Rec!*: Document the senders and receivers of messages corresponding to events in the event list (see figure 4.17).
- *Object Encyc.!*: Create entries in the object encyclopedia (see figure 4.18).

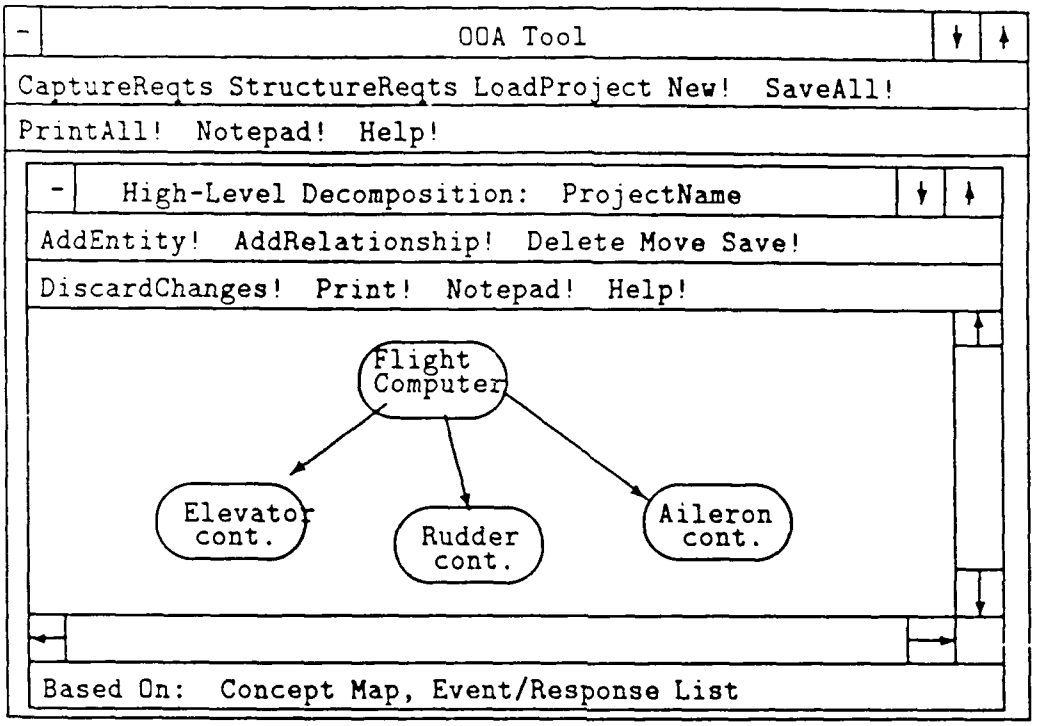
Figure 4.13. OOA Tool Storyboard: Structure Requirements Menu



When the *EID!* selection is chosen from the "Structure Requirements" menu, the External Interface Diagram window appears. This window contains commands which allow the user to draw the external interface diagram. The initial position of the window allows the user to access the main menu to get other information about the project. The bottom line of the window displays the other method steps upon which the external interface diagram is based. The scroll bar on the bottom and right hand sides of the window allow the user to scroll through the diagram with a click of the mouse button.
The following menu choices are specific to this window:

- *AddEntity!*: Adds an entity to the map. The user is prompted for a label for the entity, and is allowed to position the concept with the mouse.
- *AddLink!*: Adds a link between two entities. The user is prompted for the label to attach to the link, and is allowed to position the label with the mouse.
- *Delete*: This selection is used to delete an entity or link. After selecting this option, the user clicks the mouse on the entity or link to delete.
- *Move*: This selection is used to move an entity. After selecting this option, the user clicks on the entity to move, then drags it to its new location.
- *Save!*: Save the current external interface diagram.
- *DiscardChanges!*: Discard changes to the external interface diagram since the last save.
- *Print!*: Print the current external interface diagram.

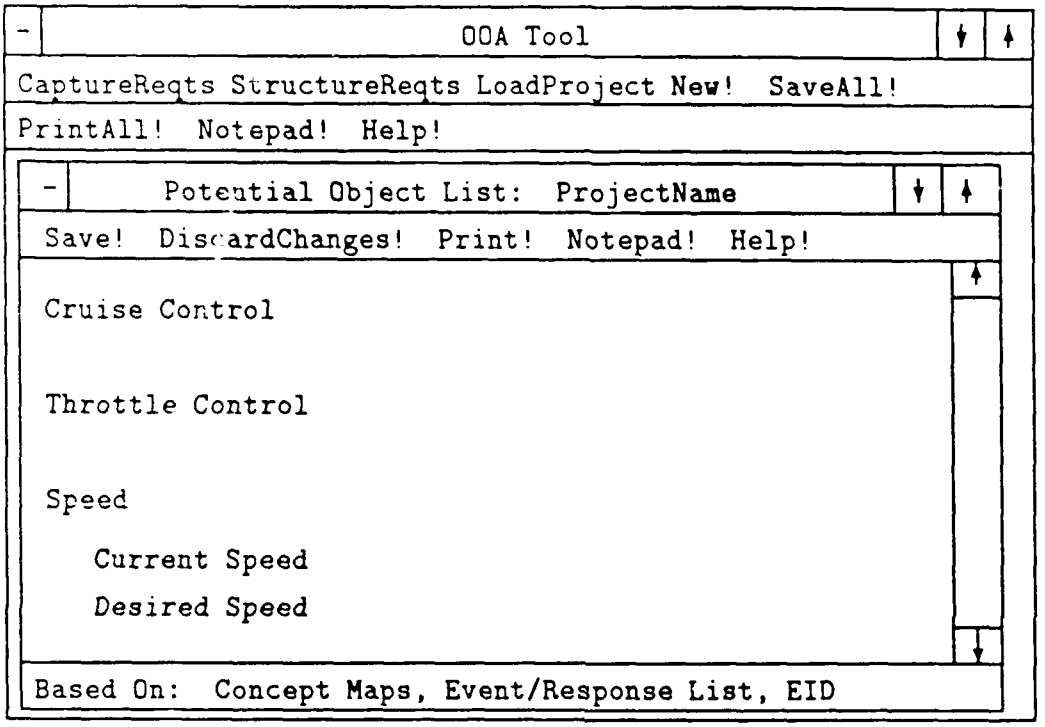
Figure 4.14. OOA Tool Storyboard: External Interface Diagram



When the *High-Level Decomp!* selection is chosen from the "Structure Requirements" menu, the High-Level Decomposition window appears. This window contains commands which allow the user to draw a structure diagram depicting the decomposition of any high-level algorithm in the system. The initial position of the window allows the user to access the main menu to get other information about the project. The bottom line of the window displays the other method steps upon which the decomposition is based. The scroll bar on the bottom and right hand sides of the window allow the user to scroll through the diagram with a click of the mouse button.
The following menu choices are specific to this window:

- *AddEntity!*: Adds a new entity to the map. The user is prompted for a label describing the entity, and is allowed to position the bubble with the mouse.
- *AddRelationship!*: Adds a relationship between two entities. The user is prompted for the label to attach to the relationship, and is allowed to position the label with the mouse.
- *Delete*: This selection is used to delete an entity or relationship. After selecting this option, the user clicks the mouse on the entity or relationship to delete.
- *Move*: This selection is used to move an entity. After selecting this option, the user clicks on the entity to move, then drags it to its new location.
- *Save!*: Save the current diagram depicting the algorithm decomposition.
- *DiscardChanges!*: Discard changes to the diagram since the last save.
- *Print!*: Print the current diagram.

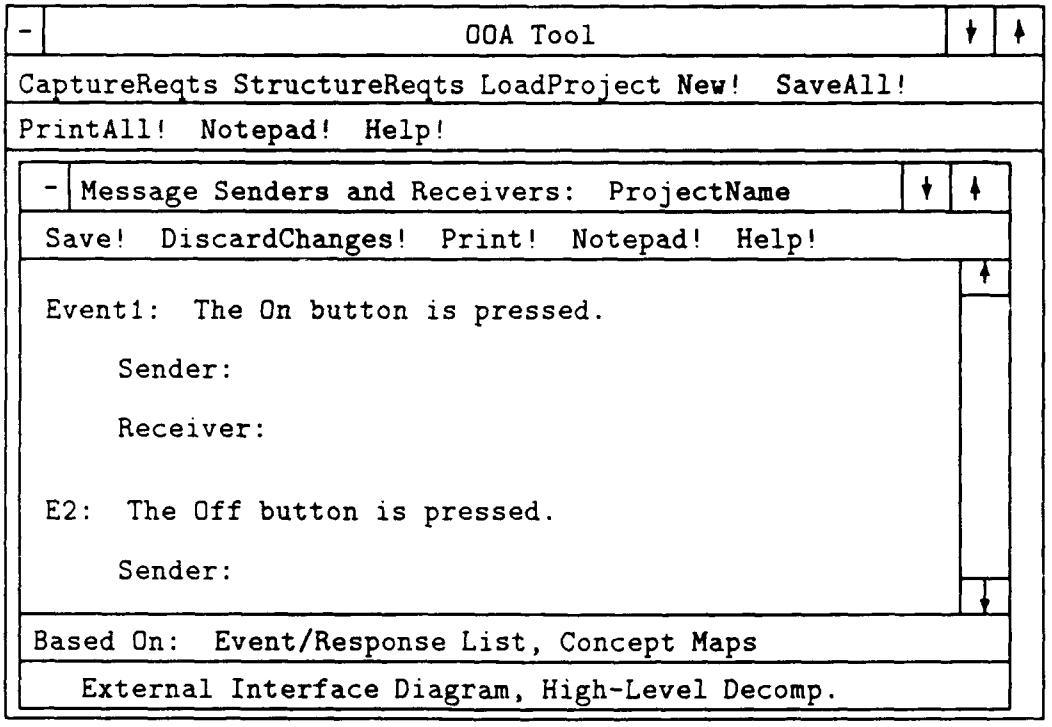
Figure 4.15. OOA Tool Storyboard: High-Level Algorithm Decomposition



When the *ObjectID!* selection is chosen from the "Structure Requirements" menu, the Object Identification window appears. This window allows the user to enter a list of potential objects. The initial position of the window allows the user to access the main menu to get other information about the project. The bottom line of the window displays the other method steps upon which the object identification is based. The scroll bar on the right hand side of the window allows the user to scroll through the list with a click of the mouse button.
The following menu choices are specific to this window:

- *Save!*: Save the current list of potential objects.
- *DiscardChanges!*: Discard changes to the list since the last save.
- *Print!*: Print the current list of potential objects.

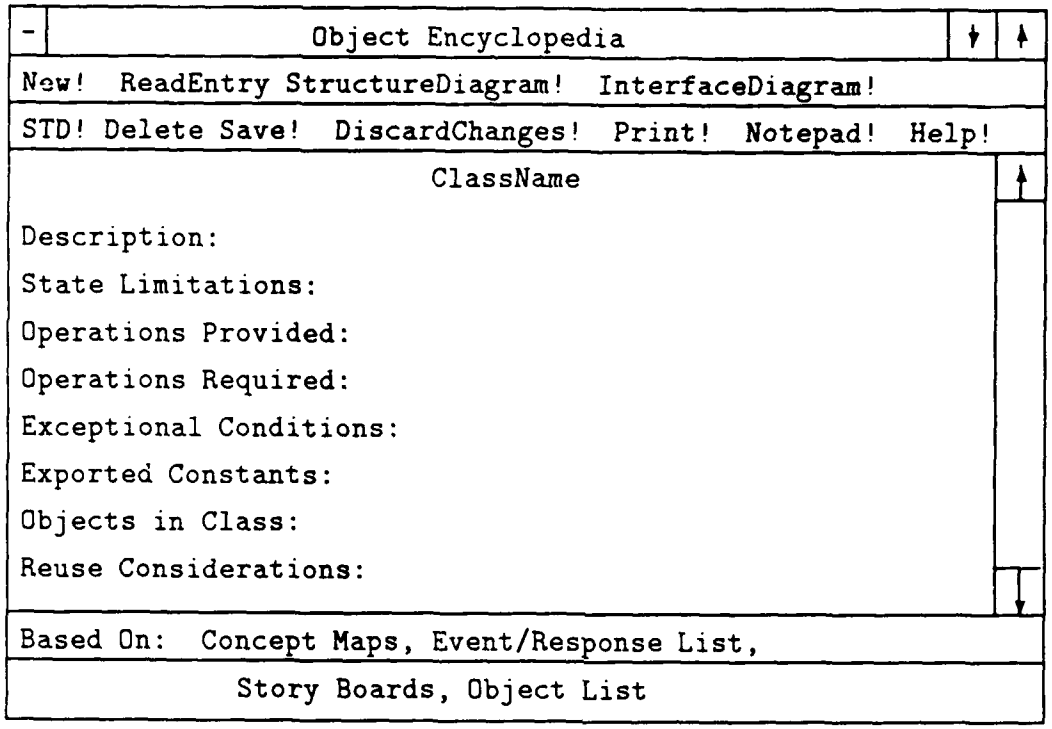
Figure 4.16. OOA Tool Storyboard: Potential Object List



When the *Msg Send & Rec* selection is chosen from the "Structure Requirements" menu, the Message Senders and Receivers window appears. This window allows the user to annotate each event in the event/response list with a name of the sender and receiver. The initial position of the window allows the user to access the main menu to get other information about the project. The bottom line of the window displays the other method steps upon which the identification of messages senders and receivers is based. The scroll bar on the right hand side of the window allows the user to scroll through the list with a click of the mouse button.
The following menu choices are specific to this window:

- *Save!*: Save the current list of message senders and receivers.
- *DiscardChanges!*: Discard changes to the list since the last save.
- *Print!*: Print the current list of message senders and receivers.

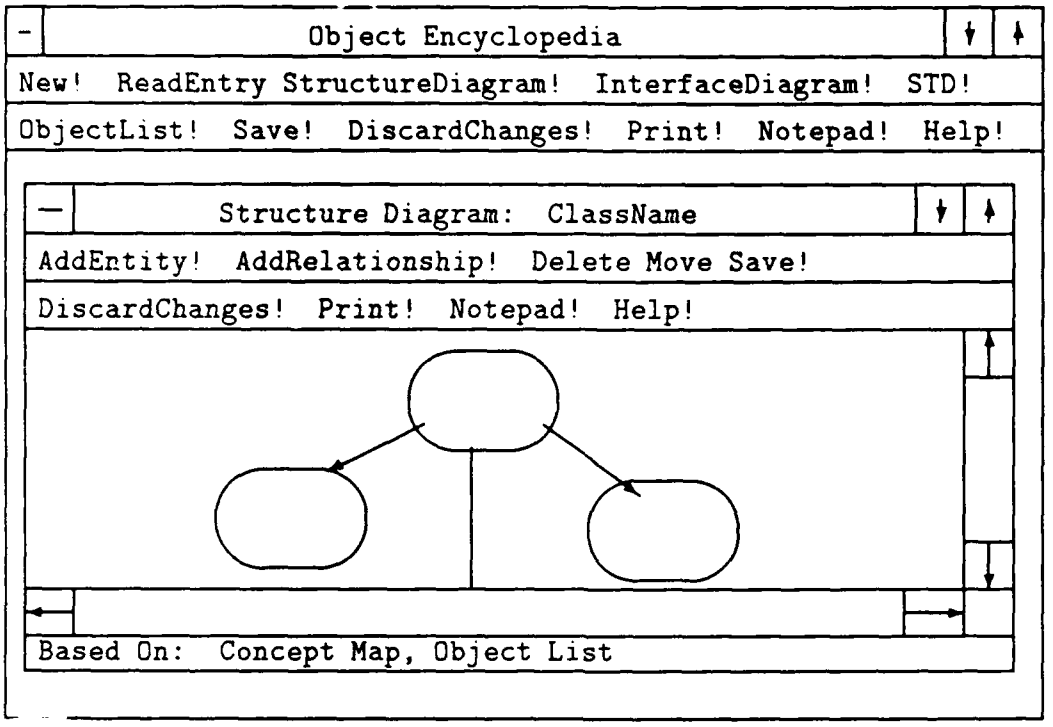
Figure 4.17. OOA Tool Storyboard: Message Senders/Receivers



When the *Object Encyc.!* selection is chosen from the "Structure Requirements" menu, the Object Encyclopedia window appears. This window contains commands which allow the user to create entries in the object encyclopedia. The window itself contains the textual information describing aspects of the class of objects.
The following menu choices are specific to this window:

- *New!*: Create a new entity to the object encyclopedia.
- *ReadEntry*: Reads an object encyclopedia entry a file.
- *StructureDiagram!*: Allows user to draw a structure diagram for the class of objects (see figure 4.19).
- *InterfaceDiagram!*: Allows user to draw an interface diagram for the class of objects (see figure 4.20).
- *STD!*: Allows user to draw a state transition diagram for the class of objects (see figure 4.22).
- *Delete*: This selection is used to delete an entry in the object encyclopedia. The user is prompted for the name of the entry to delete.
- *Save!*: Save the current object encyclopedia entry.
- *DiscardChanges!*: Discard changes to the object encyclopedia entry since the last save.
- *Print!*: Print the current entry of the object encyclopedia.

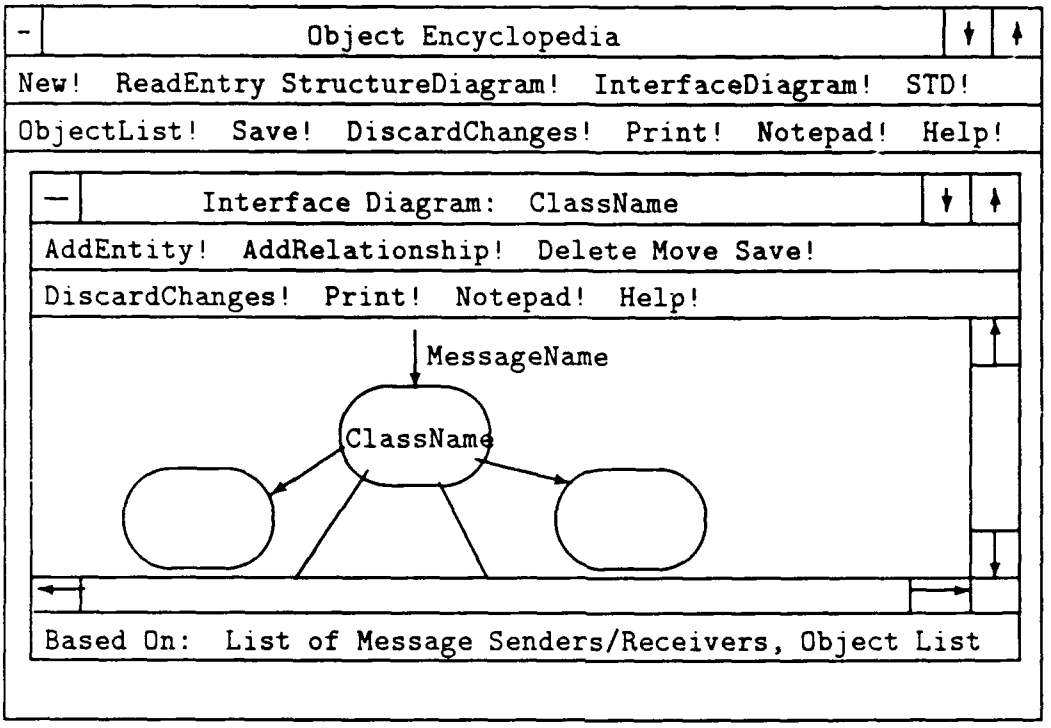
Figure 4.18. OOA Tool Storyboard: Object Encyclopedia



When the *StructureDiagram!* selection is chosen from the "Object Encyclopedia" window, the Structure Diagram window appears. This window contains commands which allow the user to draw a structure diagram for a class of objects. The initial position of the window allows the user to access the object encyclopedia menu to get other information about the class of objects. The user may click on any of the components or attributes that make up the class, and a window with the structure diagram for that class will appear. The bottom line of the window displays the other method steps upon which the structure diagram is based. The scroll bar on the bottom and right hand sides of the window allow the user to scroll through the diagram with a click of the mouse button.
The following menu choices are specific to this window:

- *AddEntity!*: Adds a new entity to the structure diagram. The user is prompted for a label for the entity, and is allowed to position the entity with the mouse.
- *AddRelationship!*: Adds a relationship between two entities. The user is prompted for the label to attach to the relationship, and is allowed to position the label with the mouse.
- *Delete*: This selection is used to delete an entity or relationship. After selecting this option, the user clicks the mouse on the entity or relationship to delete.
- *Move*: This selection is used to move an entity. After selecting this option, the user clicks on the entity to move, then drags it to its new location.
- *Save!*: Save the current structure diagram.
- *DiscardChanges!*: Discard changes to the diagram since the last save.
- *Print!*: Print the current diagram.

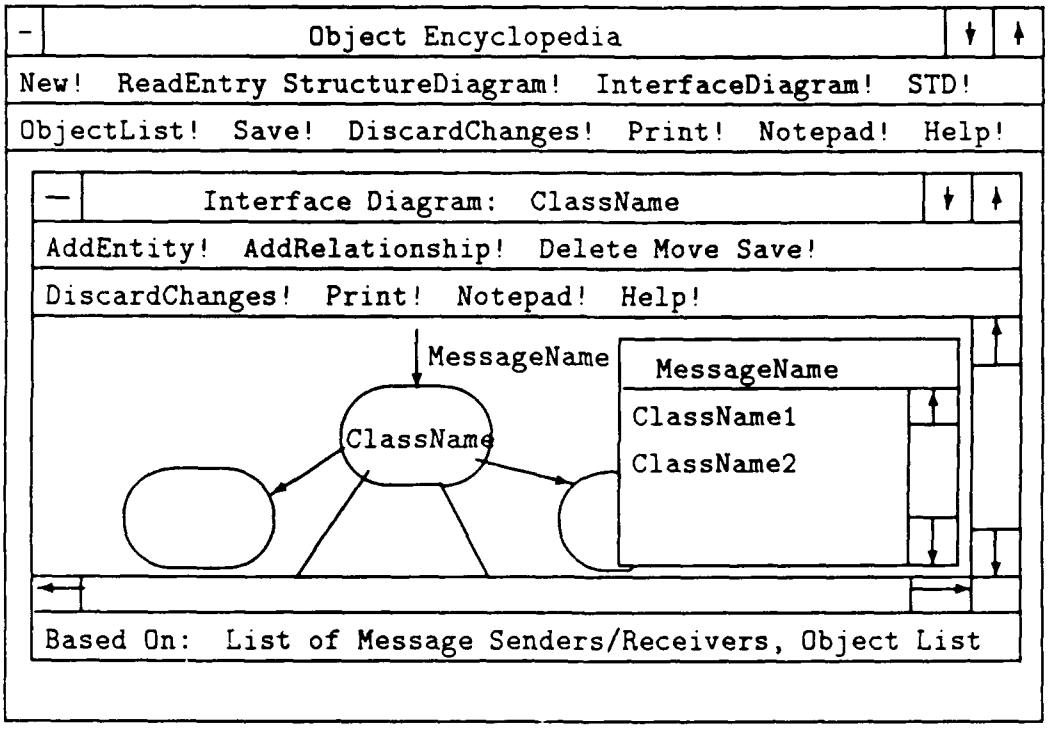
Figure 4.19. OOA Tool Storyboard: Structure Diagram



When the *InterfaceDiagram!* selection is chosen from the "Object Encyclopedia" window, the Interface Diagram window appears. This window contains commands which allow the user to draw an interface diagram for a class of objects. The initial position of the window allows the user to access the object encyclopedia menu to get other information about the class of objects. The user may click on any of the classes receiving messages from this class, and the interface diagram for the class will appear in a new window. If the user highlights the name of an incoming message, a window will open with a list of classes that send this message (see figure 4.21). The bottom line of the window displays the other method steps upon which the interface diagram is based. The scroll bar on the bottom and right hand sides of the window allow the user to scroll through the diagram with a click of the mouse button. The following menu choices are specific to this window:

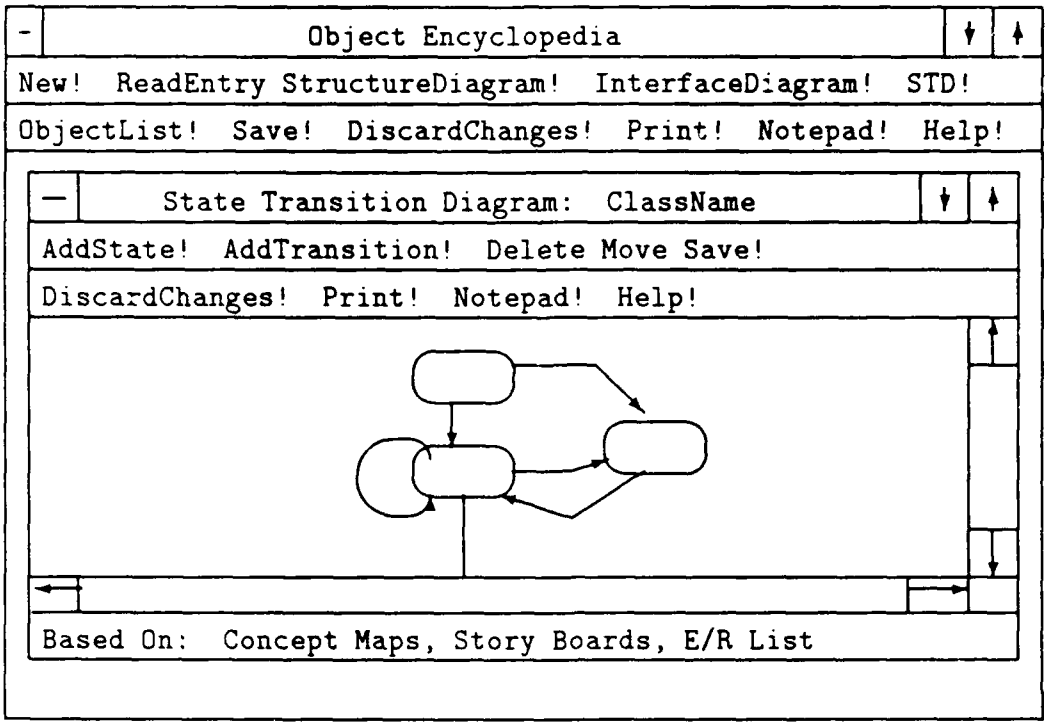
- *AddEntity!*: Adds a new entity to the interface diagram. The user is prompted for a label for the entity, and is allowed to position the entity with the mouse.
- *AddRelationship!*: Adds a relationship between two entities. The user is prompted for the label to attach to the relationship, and is allowed to position the label with the mouse.
- *Delete*: This selection is used to delete an entity or relationship. After selecting this option, the user clicks the mouse on the entity or relationship to delete.
- *Move*: This selection is used to move an entity. After selecting this option, the user clicks on the entity to move, then drags it to its new location.
- *Save!*: Save the current interface diagram.
- *DiscardChanges!*: Discard changes to the diagram since the last save.
- *Print!*: Print the current diagram.

Figure 4.20. OOA Tool Storyboard: Interface Diagram



When an incoming message is highlighted with the right mouse button on the interface diagram, a window appears with a list of classes that send the message to this class. The user may likewise highlight one of these class names and a new window will appear with the interface diagram for the class.

Figure 4.21. OOA Tool Storyboard: Highlighting Incoming Message



When the *STD!* selection is chosen from the "Object Encyclopedia" window, the State Transition Diagram window appears. This window contains commands which allow the user to draw a state transition diagram for a class of objects. The initial position of the window allows the user to access the object encyclopedia menu to get other information about the class of objects. The bottom line of the window displays the other method steps upon which the state transition diagram is based. The scroll bar on the bottom and right hand sides of the window allow the user to scroll through the diagram with a click of the mouse button.
The following menu choices are specific to this window:

- *AddState!*: Adds a new state to the diagram. The user is prompted for a label for the state, and is allowed to position the state with the mouse.
- *AddTransition!*: Adds a transition between two states. The user is prompted for the label to attach to the transition, and is allowed to position the label with the mouse.
- *Delete*: This selection is used to delete a state or transition. After selecting this option, the user clicks the mouse on the state or transition to delete.
- *Move*: This selection is used to reposition a state. After selecting this option, the user clicks on the state to move, then drags it to its new location.
- *Save!*: Save the current state transition diagram.
- *DiscardChanges!*: Discard changes to the diagram since the last save.
- *Print!*: Print the current diagram.

Figure 4.22. OOA Tool Storyboard: State Transition Diagram

The window providing access to the object encyclopedia is unique in that it takes up the entire screen. This allows more room for the sub-windows that can be opened within the object encyclopedia. Again, the user has flexibility in how he wants these windows displayed on the screen by clicking on the appropriate window commands.

The object encyclopedia window gives the analyst the following capabilities:

- Adding textual information. This is done through the main window of the object encyclopedia.
- Drawing a structure diagram for the class of objects (see figure 4.19).
- Drawing an interface diagram for the class of objects (see figure 4.20).
- Drawing a state transition diagram for the class (see figure 4.22).

A major benefit of the OOA tool, as identified at the end of the previous chapter, is to provide an ordering to the entries in the object encyclopedia. This is accomplished through the structure and interface diagrams. If the analyst clicks the mouse on a class of objects in either of these diagrams, the OOA tool will load the object encyclopedia entry for that class of objects. If the class of objects does not yet have an entry in the object encyclopedia, the user will be asked if he wants to start such an entry. This capability of clicking on objects or classes from the structure or interface diagrams allows the user to traverse the object encyclopedia entries in a hierarchical manner. The analyst can thus load the highest level object, and traverse through the diagrams to view lower level classes.

4.5 Conclusion

The concept maps and storyboards in this chapter describe a tool to assist the domain expert and/or analyst in applying the object-oriented analysis method. This tool would also provide a means of viewing the entries in the object encyclopedia in a hierarchical manner.

V. *Validation of the Object-Oriented Analysis Method*

The concepts proposed in this thesis as an Object-Oriented Analysis Method can be substantiated by applying the method to a sample analysis problem. This chapter introduces this sample problem and discusses the application of the method to the problem.

The results of applying the OOA method to the sample problem can be analyzed from a number of perspectives. The results will first be compared with the goals of the OOA method identified in chapter III. Also, the method will be compared to the results produced by two other analysis methods proposed as a precursor to OOD.

5.1 *Analysis Problem Description*

The example problem used to evaluate the OOA method is one which requires the analysis of a typical elevator control system. The problem description, taken from [Yourdon, 1989], was first used in a 1986 workshop sponsored by the Association of Computing Machinery. The following paragraphs outline the problem.

> The general requirement is to design and implement a program to schedule and control four elevators in a building with 40 floors. The elevators will be used to carry people from one floor to another in the conventional way.
>
> *Efficiency:* The program should schedule the elevators efficiently and reasonably. For example, if someone summons an elevator by pushing the down button on the fourth floor, the next elevator that reaches the fourth floor traveling down should stop at the fourth floor to accept the passenger(s). On the other hand, if an elevator has no passengers (no outstanding destination requests), it should park at the last floor it visited until it is needed again. An elevator should not reverse its direction of travel until its passengers who want to travel in its current direction have reached their destinations. (As we will see below, the program

cannot really have information about an elevator's actual *passengers*; it only knows about destination button presses for a given elevator. For example, if some mischievous or sociopathic passenger boards the elevator at the first floor and then presses the destination buttons for the fourth, fifth, and twentieth floor, the program will cause the elevator to travel to and stop at the fourth, fifth, and twentieth floors. The computer and its program have no information about actual passenger boardings and exits.) An elevator that is filled to capacity should not respond to a new summon request. (There is an overweight sensor for each elevator. The computer and its program can interrogate these sensors.)

Destination button: The interior of each elevator is furnished with a panel containing an array of 40 buttons, one button for each floor, marked with the floor numbers (1 to 40). These destination buttons can be illuminated by signals sent from the computer to the panel. When a passenger presses a destination button *not already lit*, the circuitry behind the panel sends an interrupt to the computer (there is a separate interrupt for each elevator). When the computer receives one of these (vectored) interrupts, its program can read the appropriate memory mapped eight-bit input registers (there is one for each interrupt, hence one for each elevator) that contains the floor number corresponding to the destination button that caused the interrupt. Of course, the circuitry behind the panel writes the floor number into the appropriate memory-mapped input register when it causes the vectored interrupt. (Since there are 40 floors in this application, only the first six bits of each input register will be used by the implementation; but the hardware would support a building with up to 256 floors.)

Destination button lights: As mentioned earlier, the destination buttons can be illuminated (by bulbs behind the panels). When the interrupt service routine in the program receives a destination button interrupt, it should send a signal to the appropriate panel to illuminate the appropriate button. This signal is sent by the program's loading the number of the button into the appropriate memory-mapped output register (there is one such register for each elevator). The illumination of a button notifies the passenger(s) that the system has taken note of his or her request and also prevents further interrupts caused by additional (impatient?) pressing of the button. When the controller stops an elevator at a floor, it should send a signal to its destination button panel to turn off the destination button for that floor.

Floor sensors: There is a floor sensor switch for each floor for each elevator shaft. When an elevator is within eight inches of a floor, a wheel on the elevator closes the switch for that floor and sends an interrupt to the computer (there is a separate interrupt for the set of switches in each

elevator shaft). When the computer receives one of these (vectored) interrupts, its program can read the appropriate memory mapped eight-bit input register (there is one for each interrupt, hence one for each elevator) that contains the floor number corresponding to the floor sensor switch that caused the interrupt.

Arrival lights: The interior of each elevator is furnished with a panel containing one illuminable indicator for each floor number. this panel is located just above the doors. The purpose of this panel is to tell the passengers in the elevator the number of the floor at which the elevator is arriving (and at which it may be stopping). The program should illuminate the indicator for a floor when it arrives at the floor and extinguish the indicator for a floor when it leaves a floor or arrives at a different floor. This signal is sent by the program's loading the number of the floor indicator into the appropriate memory-mapped output register (there is one register for each elevator).

Summons buttons: Each floor of the building is furnished with a panel containing summon button(s). Each floor except the ground floor (floor 1) and the top floor (floor 40) is furnished with a panel containing two summon buttons, one marked UP and one marked DOWN. The ground floor summon panel has only an UP button. The top floor summon panel has only a DOWN button. Thus, there are 78 summon buttons altogether, 39 Up buttons and 39 DOWN buttons. Would-be passengers press these buttons in order to summon an elevator. (Of course, the would-be passengers cannot summon a *particular* elevator. The scheduler decides which elevator should respond to a summon request.) These summon buttons can be illuminated by signals sent from the computer to the panel. When a passenger presses a summon button *not already lit*, the circuitry behind the panel sends a vectored interrupt to the computer (there is one interrupt for UP buttons and another for DOWN buttons). When the computer receives one of these two (vectored) interrupts, its program can read the appropriate memory mapped eight-bit input register that contains the floor number corresponding to the summon button that caused the interrupt. Of course, the circuitry behind the panel writes the floor number into the appropriate memory-mapped input register when it causes the vectored interrupt.

Summon button lights: The summon buttons can be illuminated (by bulbs behind the panels). When the summon button interrupt service routine in the program receives an UP or DOWN button vectored interrupt, it should send a signal to the appropriate panel to illuminate the appropriate button. This signal is sent by the program's loading the number of the button in the appropriate memory-mapped output register, one for the UP buttons and one for the DOWN buttons. The illumination of

a button notifies the passenger(s) that the system has taken note of his or her request and also prevents further interrupts caused by additional pressing of the button. When the controller stops an elevator at a floor, it should send a signal to the floor's summon button panel to turn off the appropriate (UP or DOWN) button for that floor.

Elevator motor controls (Up, Down, Stop): There is a memory-mapped control word for each elevator motor. Bit 0 of this word commands the elevator to go up, bit 1 commands the elevator to go down, and bit 2 commands the elevator to stop at the floor whose sensor switch is closed. The elevator mechanism will not obey any inappropriate or unsafe command. If no floor sensor switch is closed when the computer issues a stop signal, the elevator mechanism ignores the stop signal until a floor sensor switch is closed. The computer program does not have to worry about controlling an elevator's doors or stopping an elevator exactly at a level (home) position at a floor. The elevator manufacturer uses conventional switches, relays, circuits, and safety interlocks for these purposes so that the manufacturer can certify the safety of the elevators without regard for the computer controller. For example, if the computer issues a stop command for an elevator when it is within eight inches of a floor (so that its floor sensor switch is closed), the conventional, approved mechanism stops and levels the elevator at that floor, opens and holds its doors open appropriately, and then closes its door. If the computer issues an up or down command during this period (while the door is open, for example), the manufacturer's mechanism ignores the command until its conditions for movement are met. (Therefore, it is safe for the computer to issue an up or down command while an elevator's door is still open.) One condition for an elevator's movement is that its *stop button* not be depressed. Each elevator's destination button panel contains a stop button. This button does not go to the computer. Its sole purpose is to hold an elevator at a floor with its door open when the elevator is currently stopped at a floor. A red emergency *stop switch* stops and holds the elevator at the very next floor it reaches irrespective of computer scheduling. The red switch may also turn on an audible alarm. The red switch is not connected to the computer.

Target machine: The elevator scheduler and controller may be implemented for any contemporary microcomputer capable of handling this application.

The specification of this problem produced with the OOA method is included as appendix A.

5.2 *Results of Applying the OOA Method*

5.2.1 Comparison With Method Goals. The initial goals of the OOA method were outlined in section 3.1. These goals are summarized below, and examined with respect to the application of the method.

5.2.1.1 Graphical Nature Of Tools. The tools of the method should be primarily graphical, with a notation that can be understood by domain experts with little initial training.

The OOA method's communication with the domain expert is done predominately through concept maps (graphical), story boards (graphical with supporting text), and an event/response list (textual). Each of these tools is fairly unstructured, with simple, flexible notation that requires little training to understand. The fact that concept maps were used by elementary school children as a means of communicating understanding attests to their shallow learning curve [Novak and Gowin, 1984]. The text in the story boards and event/response list is presented in short statements, making it easier for a reader to follow. Not only should the domain expert be able to readily understand this material, he may well be able to prepare the items in phase I of the method himself.

The more structured *object encyclopedia* entries will still be reviewed by the domain expert(s), and thus need to be easily understood. The contents of the entries rely heavily on the graphical structure diagram, interface diagram, and state transition diagram. The syntax of these diagrams is similar to that of the concept map, so they too should allow the reader to concentrate more on the meaning of the diagram rather than the syntax.

5.2.1.2 Ease of Application. The method should be straightforward in its application.

The OOA method includes some heuristics to apply the steps of the method.

Once the initial information is gathered from the domain expert (phase I of the method), each of the steps taken to structure the information and create *object encyclopedia* entries is based on information present from phase I.

One of the more difficult aspects of applying the OOA method is the necessity to iterate between some of the steps in the method. For example, in the application of the method to the elevator problem, the similarity between the UP and DOWN request panels and each elevator's control panel was not evident until these classes were documented as entries in the object encyclopedia. Recognition of their parallel characteristics required the revision of the preliminary object list (page A-30) and the *object encyclopedia* entry for the control panel (page A-48).

5.2.1.3 Identification of Objects, Attributes, and Object Interaction. The method should support the identification of objects, their attributes, and the interaction among objects. This necessarily includes documenting the object's external interface.

The method supports the identification of objects and their attributes from the concepts on the domain expert's concept maps. The identification of the interaction among objects, or messages passed between them, is supported by the method via the event/response list, as well as the linking verbs on the concept maps. The objects, attributes, and interaction among objects is further documented for each class of objects in the series of entries in the *object encyclopedia.*

The key to identifying the objects and messages in the OOA method is in the construction of the concept maps, story boards, and event/response list in phase I. This phase may require a considerable effort to document and refine the requirements in order to adequately specify the problem.

In the application of the OOA method to the elevator problem, the vast majority (66 out of 75) of the concepts in the initial concept maps turned out to be directly related to objects or attributes documented in phase II of the method. Likewise, all

of the event/response pairs in the event/response list resulted in at least one message sent between object classes.

5.2.1.4 Top-Down Nature of Model. The model of the system should be presented in a top-down hierarchical manner.

As stated in chapter III, the levels of hierarchy in the description of the system can be seen through the interface and structure diagrams in the *object encyclopedia*. As seen in the description of the elevator control system, the entries in the *object encyclopedia* are ordered by first describing the main object class (*Elevator Control System*), then its component classes (*Elevator*, and *Control Panel*), before defining the remaining classes (in alphabetical order).

The ordering of classes in the *object encyclopedia* is complicated by the nature of object-oriented systems; there is not always a clear distinction between levels of abstraction, and these levels are not as neatly organized into a tree structure familiar in a functional description of a problem. Often, a lower level class (such as *Weight* or *Address* in the elevator problem) is used by a variety of other classes in the system. Also, an object of a certain class may send messages to other objects at different levels of abstraction. For example, the interface diagram for the *Control Panel* class (see figure A.30) shows that a control panel object may send messages to higher level objects (via the *Button Pushed* message), or lower level objects (e.g. *Address*, or an *Input Register*). To further compound the problem, objects of the *Control Panel* class may appear at different levels of abstraction. The UP and DOWN Request Panels are logical components of the overall *Elevator Control System*, while an elevator control panel is a component of the *Elevator* class. These complexities make it difficult to hierarchically order the descriptions of the object classes in the *object encyclopedia*. It was these complexities which prompted the idea for the traversal of classes through the structure and interface diagrams via the OOA tool described in chapter IV.

5.2.1.5 Support for Large, Embedded Systems. The method should support the definition of large embedded systems.

The application of the OOA method to the elevator problem demonstrates the ability of the method to document the requirements for an embedded system. Indeed, the event/response list, frequently used in the analysis of embedded systems, proved effective in identifying messages between objects in the OOA method. However, few software engineers would consider the elevator problem, though not trivial, to be a "large" embedded system. Larger problems require the ability of the analyst to present the details of the system in a hierarchical manner. Though not proven decisively in the example included here, the OOA method and tool has the ability to provide such hierarchical structuring, as discussed in the previous section.

5.2.1.6 Minimal Redundancy. Different method representations should include minimal redundancy between them.

The second phase of the OOA method contains a great deal of the information captured in phase I. However, this is expected since one phase follows from the other, with the different phases aimed at communication with different audiences. Potentially harmful redundancy occurs when different views within the same phase of the method needlessly contain overlapping information. Some redundancy exists in all methods where the same entity is viewed from multiple views. This redundant information is difficult to keep current when changes are made to one view of the entity.

In phase I of the OOA method, an indication of a message between objects may show up in each of the event/response list, story boards, and action verb links on the concept maps. However, each of these views contains different information about the messages. The event/response list describes the message as well as any response that is undertaken as a result of receiving the message. The concept map describes the message in terms of its relationship with its sender and receiver. The

story boards place the message in context with other messages and the state of the entities in the system. Thus, while the existence of a message may be known from different views, each of these views contains different information about the message.

In phase II of the OOA method, there is some redundancy in the information provided on the interface diagrams. The messages that show up leaving one entity must show up in the interface diagram of the receiving entity. For example, the *Direction Of* message from the *Elevator Control System* class to the *Elevator* class must show up on the interface diagram (figure A.24) and messages sent list (page A-36) of the *Elevator Control System*, as well as the interface diagram (figure A.27) and messages received list (page A-41) of the *Elevator* class. However, the repetition is inevitable if the requirements specification is organized such that the description of each class is localized. This organization makes it easier to map the analysis into an object-oriented design.

Thus, while there is some replication in the information contained in different diagrams in phase II, this redundancy is minimal and is caused by attempts to satisfy other goals. The redundancies that do exist can be exploited by the OOA tool presented in chapter IV to check the consistency of the object classes with each other.

5.2.1.7 Mapping Into OOD. The result of the method should map cleanly into a "Booch-flavored" object-oriented design.

Viewing the entries in the *object encyclopedia* for the elevator problem, the mapping into an object-oriented design is fairly obvious. Each documented class of objects in the *object encyclopedia* (except for the external entities that have no software driver) will likely show up as objects in the design. Also, each of the documented classes contains enough information about the class's behavior and external interface to design the class.

5.2.2 Comparison With Other Analysis Approaches. The "elevator problem" has been used to illustrate other analysis tools, including [Yourdon, 1989] and [EVB, 1989]. This enables a comparison of the OOA method to these methods as a precursor to object-oriented design.

5.2.2.1 Modern Structured Analysis. Although Yourdon makes no claims as to the applicability of *Modern Structured Analysis* as a precursor to object-oriented design, Booch maintains that data flow diagrams (even developed with a functional approach) are appropriate to capture the problem space for an object-oriented design [Booch, 1986:212]. Also, the *abstraction analysis* process proposed by [Seidewitz and Stark, 1987] begins with the data flow diagrams produced during structured analysis.

The high level data/control flow diagrams for the elevator problem, taken from [Yourdon, 1989], are shown in figures 5.1 through 5.4. These diagrams provide a feel for how the problem is modeled using *Modern Structured Analysis.*

Looking at the these figures, it becomes readily apparent that the data/control flow diagrams do not map well into an object-oriented design. While some of the lower-level objects can be discerned from the data stores, it is more difficult to identify the mid-level objects. Furthermore, the messages sent and received by each object are not obviously grasped from a single data flow diagram. A significant amount of additional effort must be applied to transform these data/control flow diagrams (as in [Seidewitz and Stark, 1987]) into a specification appropriate for an object-oriented design.

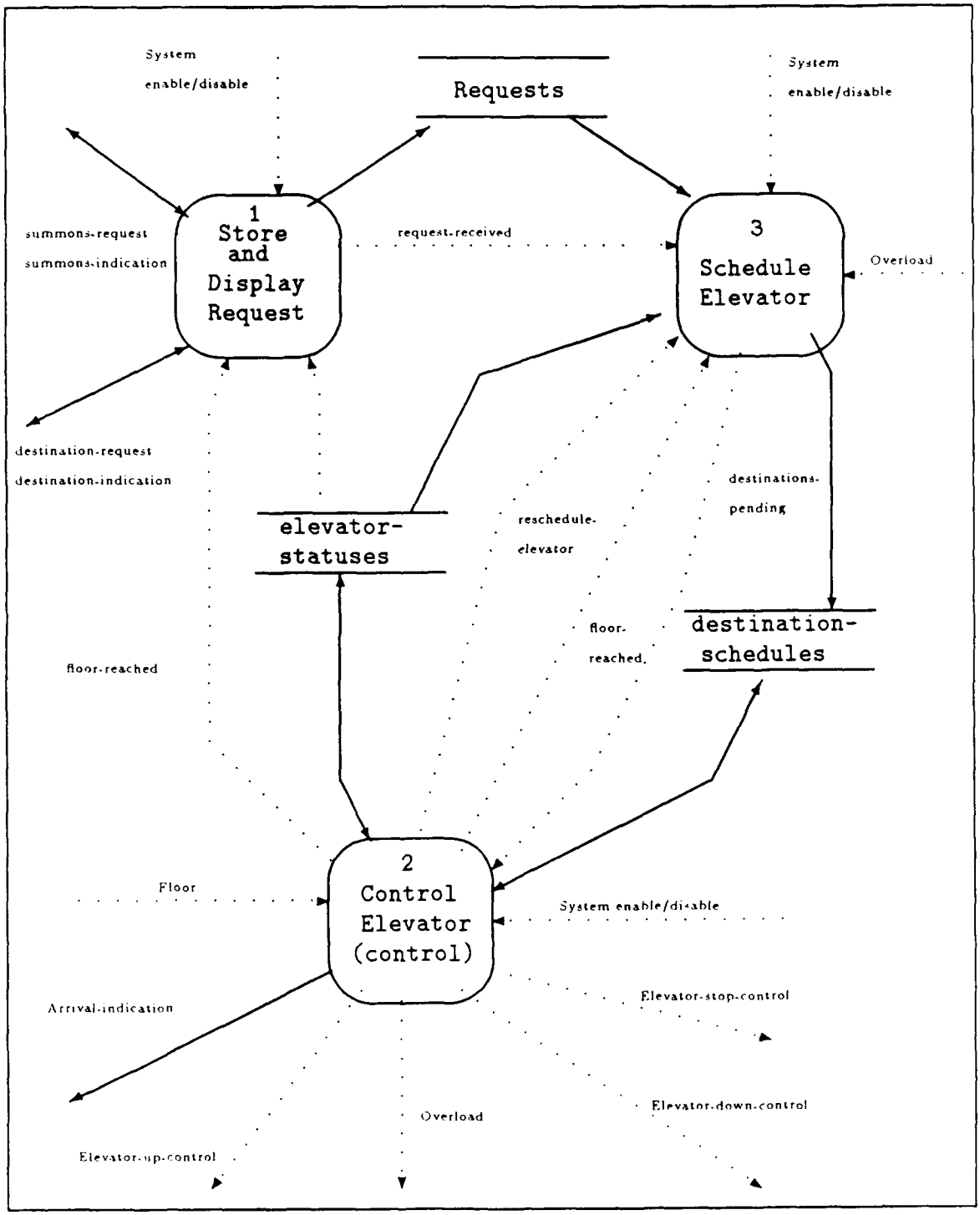


Figure 5.1. Schedule and Control Elevator: Elevator Essential Model [Yourdon, 1989:638]

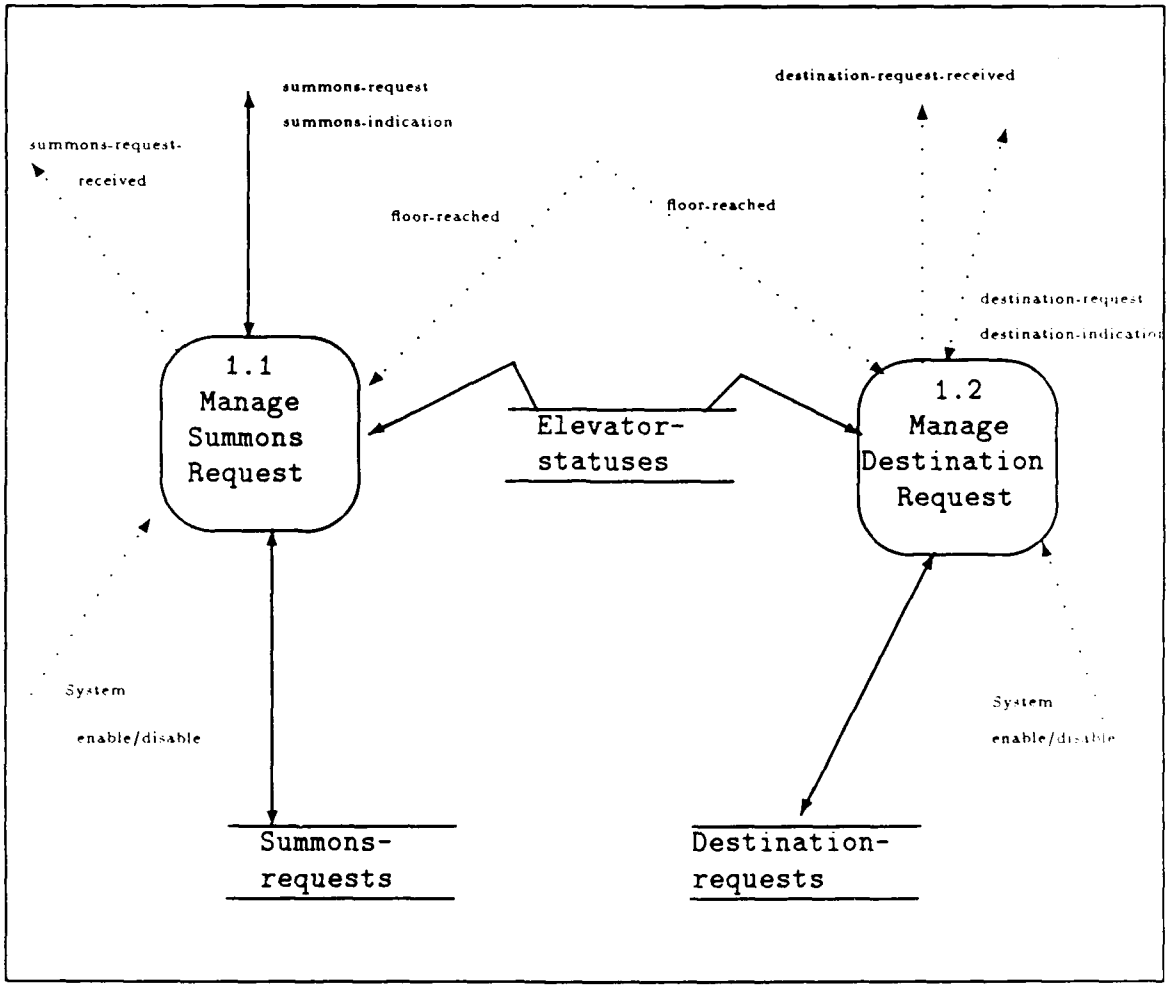


Figure 5.2. Store and Display Request [Yourdon, 1989:639]

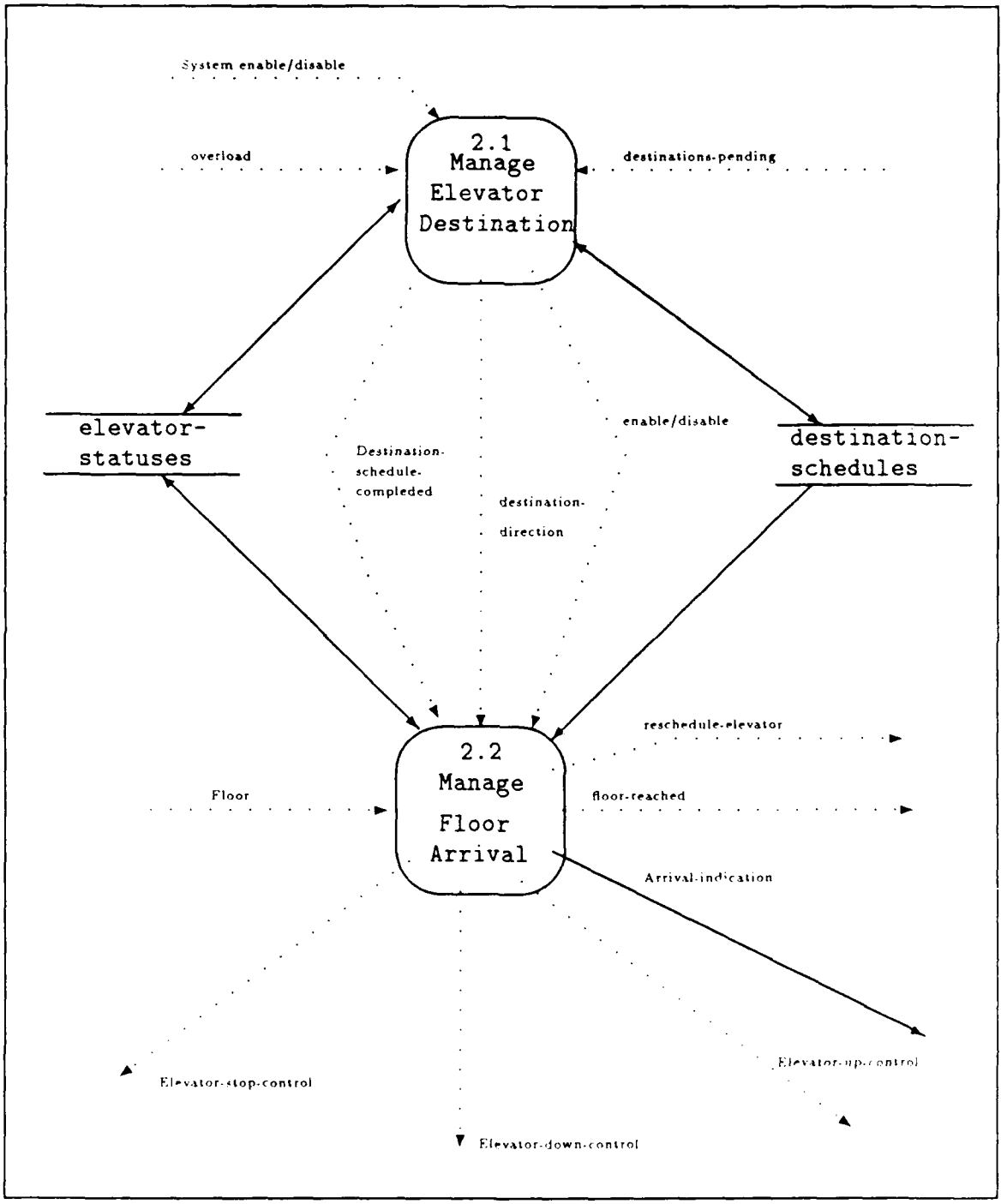


Figure 5.3. Control Elevator [Yourdon, 1989:613]

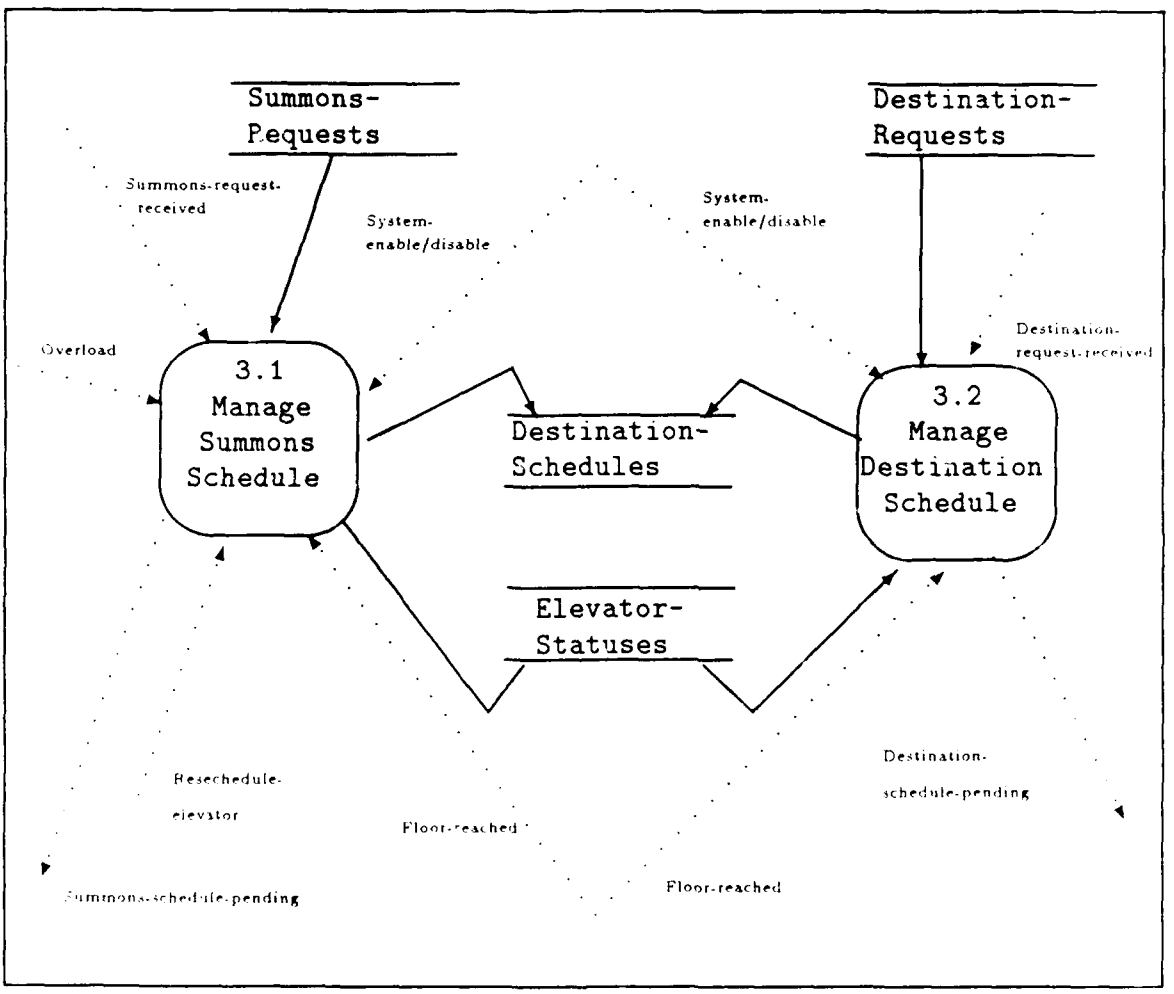


Figure 5.4. Schedule Elevator [Yourdon, 1989:619]

5.2.2.2 EVB's Object-Oriented Requirements Specification. The OOA method presented in this thesis produces a model with some similarities to EVB's Object-Oriented Requirements Specification (see section 2.3.2.4). The closest resemblance comes in the use of a multi-dimensional view of each class of objects. The *object encyclopedia* entries presented here share many common characteristics with EVB's *object class specification (OCS).*

Despite the similarities between the two specifications, there are some fundamental differences in two approaches. First, the EVB approach provides few guidelines for identifying object classes and developing the set of object class specifications. EVB provides some "hints" for starting an object-oriented requirements analysis, both from scratch and from an existing set of requirements, but these "hints" are not developed into a set of specific steps and heuristics to guide the analyst. The steps of OOA method presented in this thesis attempt to provide the analyst with a more deliberate approach to modeling the system requirements.

The OOA method recognizes a need for communication with both domain experts and designers. The *object encyclopedia* entries (and EVB OCSs) are structured representations aimed at presenting information to the designer. The OOA method also includes valuables tools for communication with the domain expert(s). The concept maps, story boards, and event/response list provide a more *unstructured* representation of the system which is often easier for the domain expert to follow (and construct), and also serves as a basis for the more structured models.

The organization of the OCSs in EVB's approach is based primarily on the class's potential for reuse. While the method of this thesis recognizes the potential of reusable components, it places more consideration in the ordering of class descriptions based on the system hierarchy, especially through the use of the OOA tool.

Finally, the OOA method contains an interface diagram in the *object encyclopedia* entries. This diagram, lacking in the EVB OCS, provides a graphical view of

the class in terms of the messages sent to and received by other classes. This diagram can provide a valuable insight at a glance into the interaction among object classes in the system.

5.3 Conclusion

The application of the object-oriented analysis method on the elevator problem demonstrates the viability of the method in analyzing the requirements of an embedded software system. The combination of the concept maps, story boards, and event/response list captures both the problem elements and their interaction, in an unstructured format readily understandable by the domain experts. In addition, the method's structuring of this information into *object encyclopedia* entries provides a more straightforward mapping into an object-oriented design than a more traditional (functional) analysis method.

VI. Conclusions and Recommendations

The final chapter of this thesis begins with a summary of the goals and objectives which guided this research. Next, conclusions are presented based on the investigation into and application of object-oriented analysis techniques. The chapter concludes with a list of areas recommended for further research following from this study.

6.1 Summary

The main goal of this thesis was to develop an Object-Oriented Analysis (OOA) method to model software requirements as a precursor to object-oriented design. The research was in part inspired by the work of [Barnes, 1988], who recommended the use of the concept map as a tool to replace the *informal strategy* as a representation of the problem space.

This thesis was guided by the series of objectives outlined in chapter I. The first objective was to determine the requirements of an object-oriented analysis method in terms of the information the method should capture. These requirements were gathered by reviewing the existing literature. The topics covered by this examination included the application of object-oriented techniques to the coding and design phases of the life cycle, as well as various approaches to requirements analysis.

The next objective was to define the steps which would specify the OOA method. These steps were defined by first selecting the tools needed to represent the information deemed appropriate during OOD, as identified in the literature review. The OOA method viewed the requirements analysis process as a bridge of communication between domain experts and a designer. In light of this, the method tools were selected to be straightforward in syntax (so that domain experts wouldn't be overwhelmed with unfamiliar symbols), yet structured in final form (so that the designer wouldn't be overwhelmed with problem complexity). Concept maps, story boards,

and an event/response list capture the software requirements from the domain expert. A series of entries in the *object encyclopedia* for each class of objects serves to communicate the analysis to the designer. The specific method steps evolved, through application on various sample problems, to traverse from one representation to the other.

The OOA method was then examined to see how it could benefit from automated support. Decision Support System (DSS) concepts were applied to define the nature of a tool to support the OOA method. The greatest areas of potential support would be the tool's ability to traverse the hierarchy of *object encyclopedia* entries, and perform consistency checks on the model of software requirements.

The final objective was to validate the concepts of this research by applying the OOA method to a sample problem. The method was evaluated with respect to the initial method goals, and the results of the analysis compared to those obtained through other methods.

6.2 Conclusions

The concept map is a viable tool for identifying objects and classes of objects in the problem space. The informal syntax of the concept map makes it ideal for communication between an analyst and a domain expert. However, it is not sufficient for specifying the entire problem. The concept map is useful for showing structural relationships among objects, but is weak for describing the dynamic interaction among objects. Therefore, the concept map must be used in conjunction with other tools (such as story boards and the event/response list) to fully capture the breadth of information required during object-oriented design.

The complex nature of an object requires a sophisticated set of tools to represent the information needed to design it. An object contains elements from both the information and functional domains, and cannot be fully characterized by one view alone. Traditional functional analysis tools, such as the data flow diagram,

are therefore not sufficient in themselves to adequately represent objects since they concentrate on only one dimension. Furthermore, even when these tools are used in combination with other tools which capture the missing dimension, they are often unclear to the designer because they lack a one-to-one mapping between objects and entities on the diagram.

Perhaps the most significant innovation resulting from this thesis is the use of the event/response list in identifying messages passed between objects. This vehicle provides a description of the interaction of the software to its external environment. Each external event in the list (excluding periodic events) corresponds to a message passed to a particular object. The responses to these events suggest the existence of additional messages passed between objects.

A somewhat surprising conclusion is that the functional and object-oriented views may not be as detached as Booch seems to imply. At the highest levels of abstraction, an actor object's state may be composed entirely of the states of its component objects. At this level, it becomes difficult to distinguish between an actor object and a process. Often, the distinction is more one of nomenclature rather than substance, as in the difference between an "elevator scheduler" object and a "schedule elevator" process.

Finally, an interesting observation resulting from this research deals with the topology of an object-oriented design verses that of a structured design. The result of applying structured design is typically a tree-shaped network of modules in a structure chart. The set of sub-modules which compose the parent module (those which stem from it on a structure chart) is identical to the set of modules the parent calls. On the other hand, the topology of an object-oriented design is more like that of a directed graph. In an object-oriented design, the set of objects receiving messages from a particular object is potentially greater than the set of sub-objects which compose that object. In other words, unlike a module in a structured design, it is not unusual for an object to interact with other objects at the same (or even

higher) level of abstraction. This characteristic makes it more difficult to distinguish between levels of abstraction in an object-oriented design than in a structured design.

6.3 Recommendations

First and foremost, the OOA tool should be designed and implemented to further apply and test the OOA method developed in this thesis. The availability of a software tool to assist in the application of the method not only would make it easier to apply the OOA method, but would also facilitate the analysis of larger systems. With this experience, further evaluation of these concepts can be undertaken. The description of the tool presented in chapter IV provides a basis for the design and implementation of the tool.

A more comprehensive evaluation of the OOA method should be performed. This thesis demonstrated an approach to an object-oriented analysis of the problem space that intuitively seems more appropriate for use with OOD. However, the true benefit of a good approach to requirements analysis is not seen until later in the project life cycle, when analysis errors can have a great impact on the quality of the software. Further evaluation should compare the results of this method to that of other methods, not only during the analysis phase, but throughout the project's life cycle.

More research needs to be done into the practical aspects of applying object-oriented design to real-time embedded systems. Issues such as interrupt handling and concurrency are not emphasized by Booch in his definition of object-oriented design. The presence of distributed processing in many such applications makes concurrency an especially important issue. To date, few guidelines exist to assist a designer in introducing concurrency into an object-oriented design. This may be a result of the limitations of traditional object-oriented languages (e.g. Smalltalk) which lack features to manage concurrent processing. The existence of tasking in Ada, however, enables the implementation of concurrent object-oriented programs. Potentially, all

objects in a system could execute concurrently; however, efficiency considerations impose limits on this strategy. Further research needs to be undertaken to provide heuristics for employing concurrency in the object-oriented paradigm.

An interesting observation during the course of this research calls for further investigation. In Booch's definition of object-oriented design as it applies to Ada, an object presents a uniform interface to all who have visibility to it. Objects may receive the same messages from other objects, whether they are at higher, lower, or the same level of abstraction. However, the information hiding principle seems to warrant a different view of the object from different levels of abstraction. Further research can provide an insight into what avenues, if any, should be taken to give an object greater flexibility in presenting its interface to different levels of abstraction.

One of the assumptions of the OOA method was that the domain expert has previously defined the software requirements; the analysis method only captures those requirements and adds structure to them for the design activity. Further research needs to be done in the psychological and procedural aspects of constructing the initial concept maps, story boards, and event/response lists to define a complete set of requirements.

Finally, the issue of reusability needs to be addressed. One of the potential benefits of the object-oriented paradigm is in the reuse of object classes. After all, the set of predefined classes is one of the key strengths of object-oriented languages such as Smalltalk and Actor. The issue of reuse seems to warrant application at the organizational level as well as the project level. Clearly, the *search* for reusable components should be undertaken at the project level, during the analysis or design activities. However, the decision to expend the additional resources required to design a component to be reusable is not as clear—this decision is as much managerial as technical, and is influenced by the requirements of other projects in the organization. Currently, the *object encyclopedia* entries contain an assessment of the class's reuse potential. The advancement of a more complete framework for identifying,

designing, and applying reusable object classes needs to be investigated.

6.4 Closing Remarks

The successful application of object-oriented design depends on a complete model of software requirements. The object-oriented analysis method presented in this thesis constructs such a model. Furthermore, the information contained in this model is structured around the objects in the problem space. This organization furnishes a more straightforward mapping into OOD than functional analysis methods such as *Structured Analysis*. The OOA method also provides more guidelines and structure for the designer than the *informal strategy* method originally proposed by Booch, while striving to maintain a more unstructured, graphical means of communication with the domain experts. The OOA method presented here yields a basis for application and further study in object-oriented requirements analysis, object-oriented design, and the proper use of Ada language constructs.

Appendix A. *Analysis of an Elevator Control System*

A.1 Purpose of Elevator Control System

The purpose of the elevator control system is to schedule and control four elevators in a building with 40 floors. The elevators will be used to carry people from one floor to another in the conventional way.

The elevator control system receives signals from summons panels on each floor, and command panel buttons and floor sensors associated with each elevator. It controls the movement of the elevators and the setting of lights on the summons panels and elevator control panels.

The elevator control system will only control the movement of the elevators from floor to floor. The computer program does not have to worry about controlling an elevator's doors or stopping an elevator exactly at a level (home) position at a floor. The elevator manufacturer uses conventional switches, relays, circuits, and safety interlocks for these purposes so that the manufacturer can certify the safety of the elevators without regard for the computer controller. For example, if the computer issues a stop command for an elevator when it is within eight inches of a floor, the conventional, approved mechanism stops and levels the elevator at that floor, opens and holds its doors open appropriately, and then closes its door. If the computer issues an up or down command during this period (while the door is open, for example), the manufacturer's mechanism ignores the command until its conditions for movement are met.

A.2 Concept Maps

The concept maps on the following pages describe the components of the elevator control system.

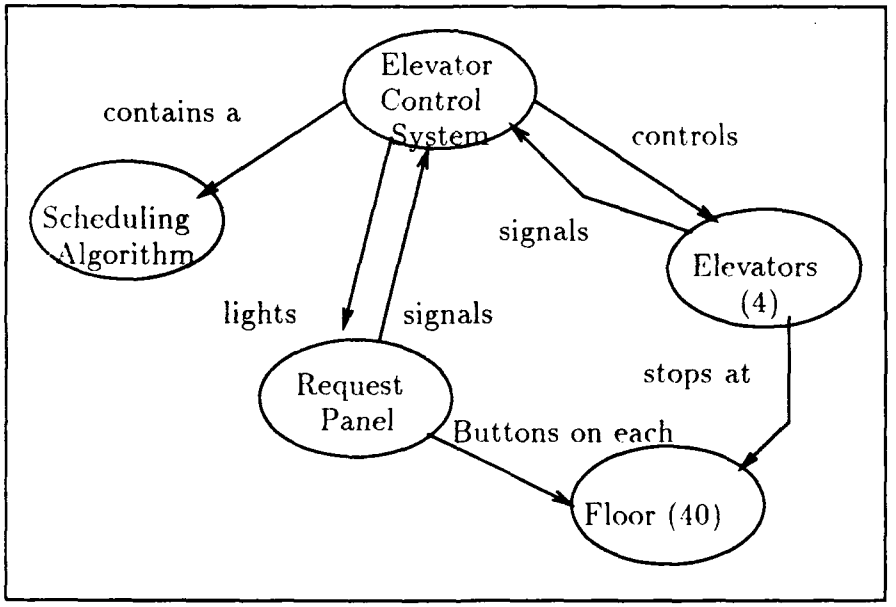


Figure A.1. Overall Elevator Control System

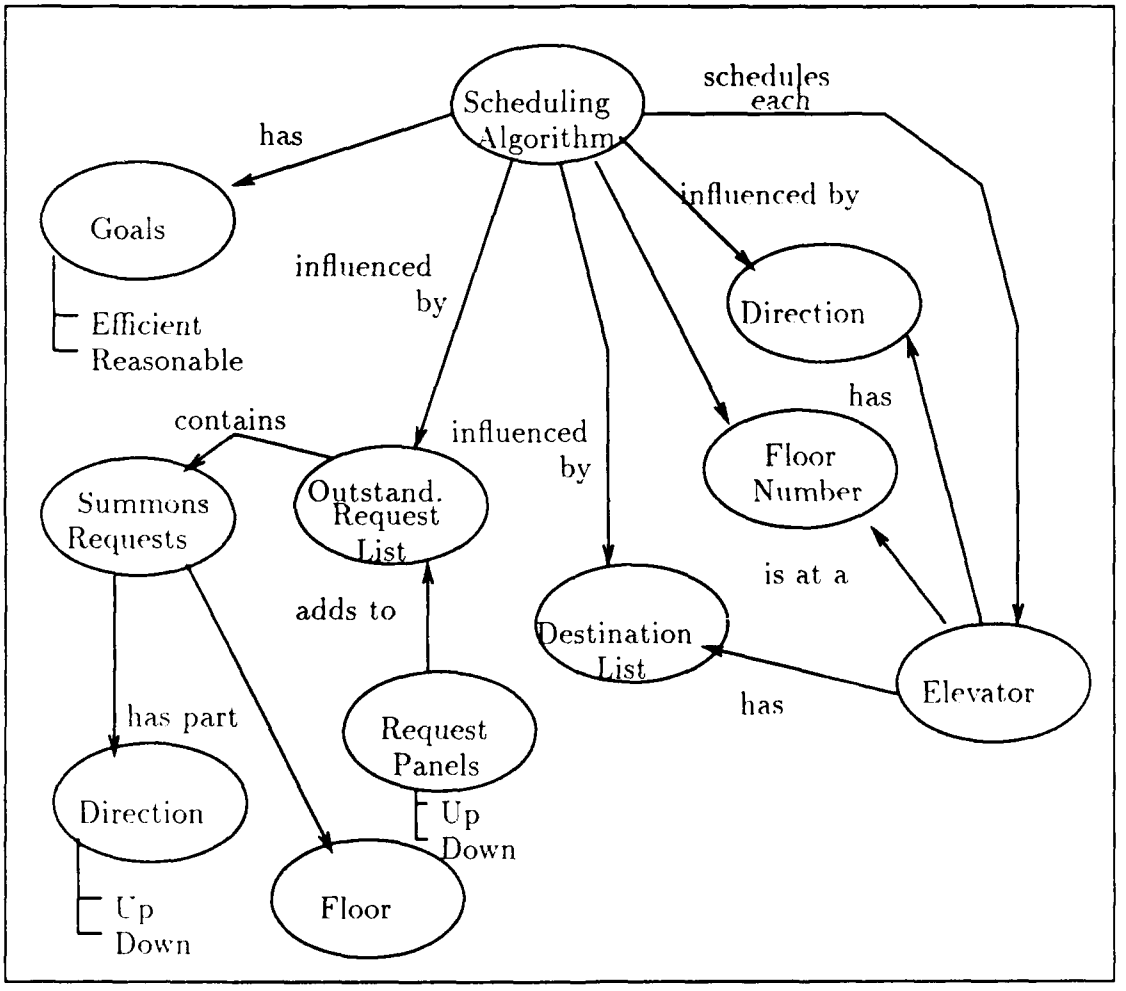


Figure A.2. Scheduling Algorithm

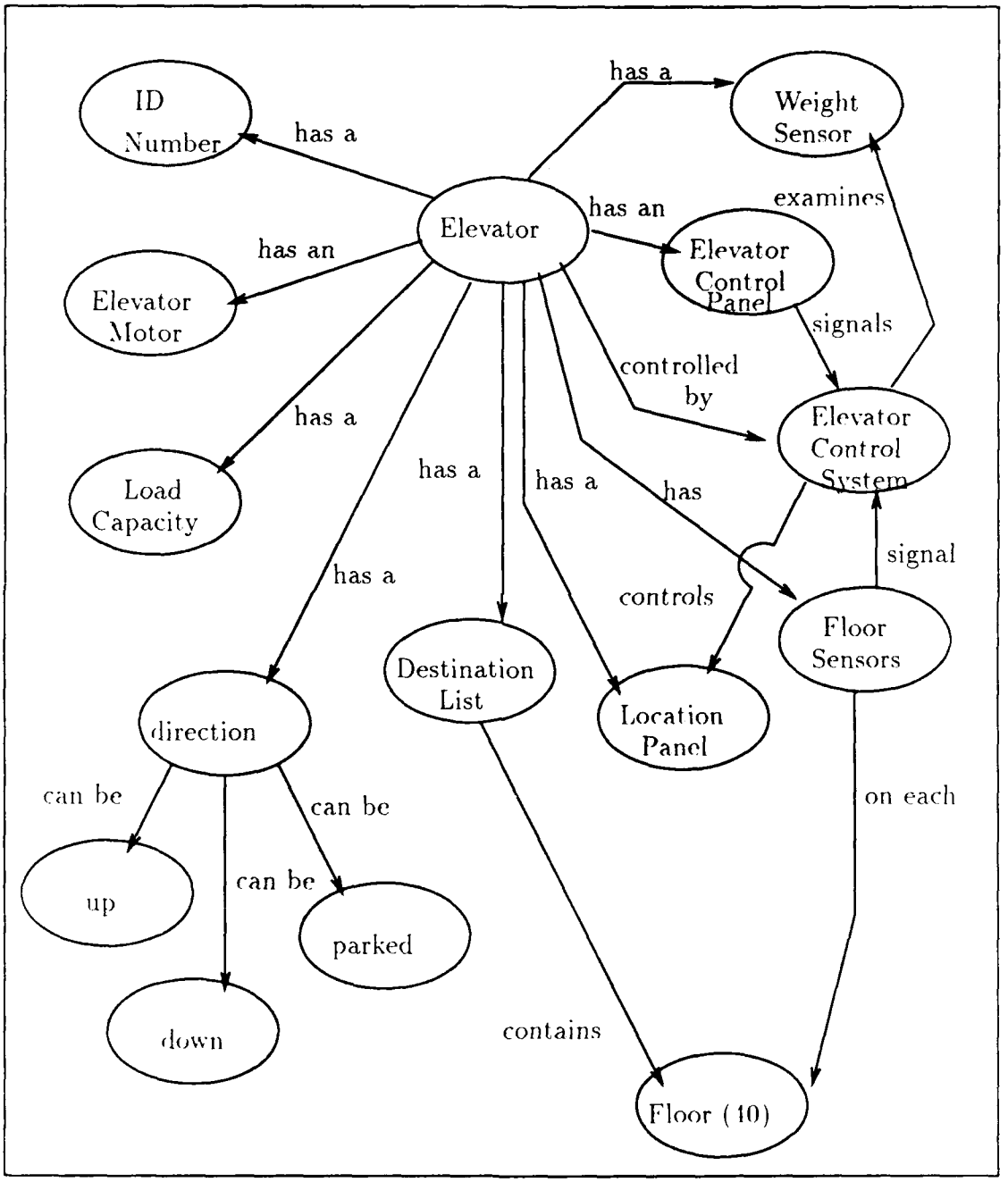


Figure A.3. Elevator Components

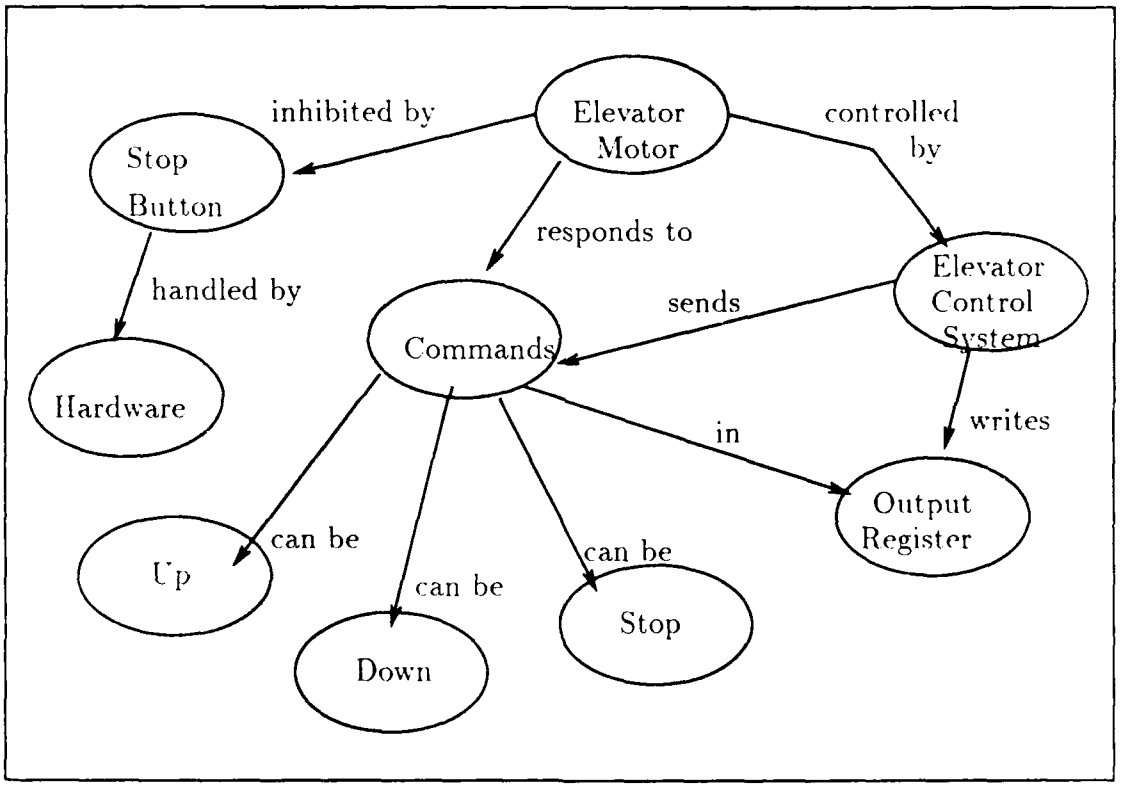


Figure A.4. Elevator Motor

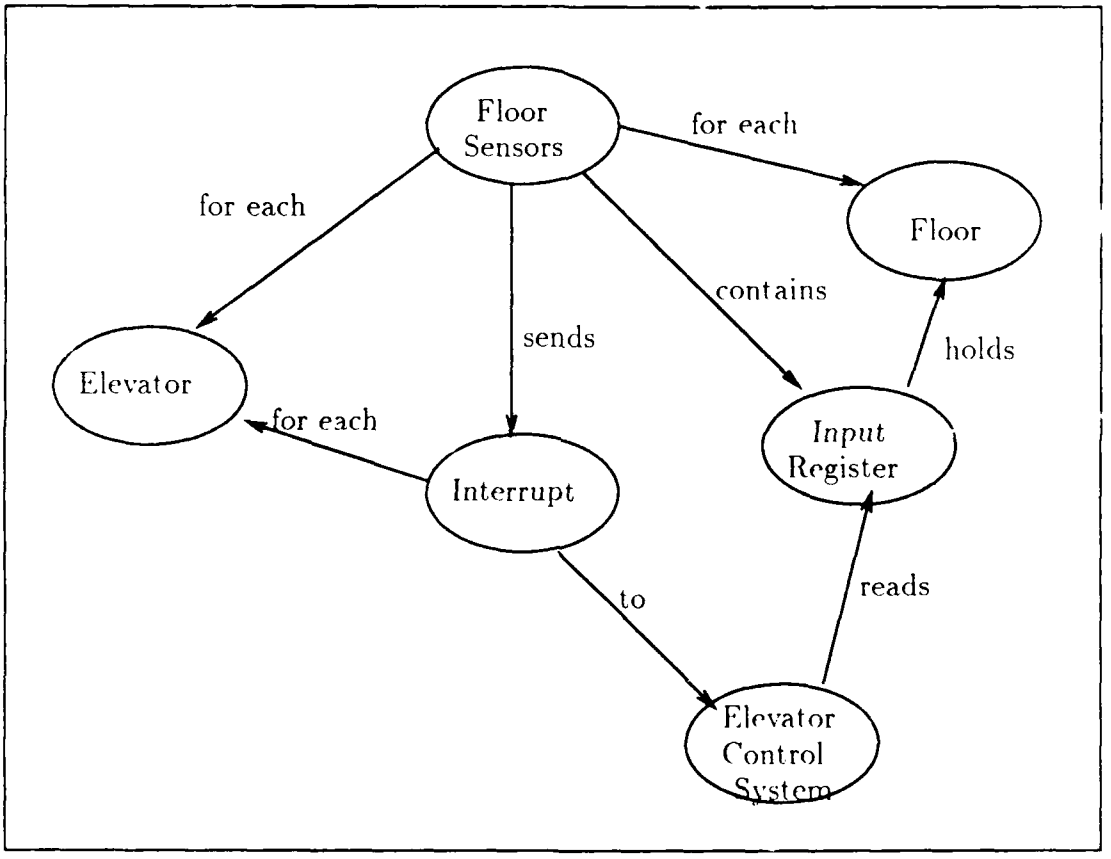


Figure A.5. Elevator Floor Sensors

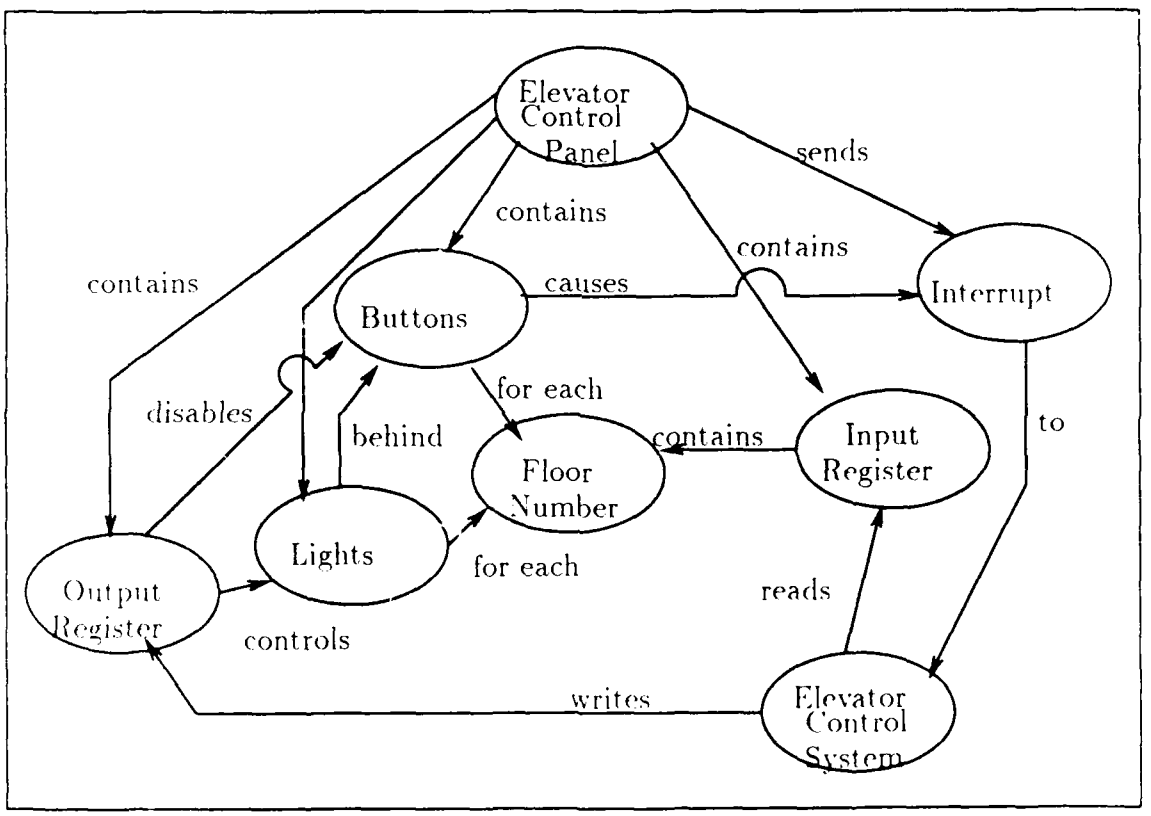


Figure A.6. Elevator Control Panel

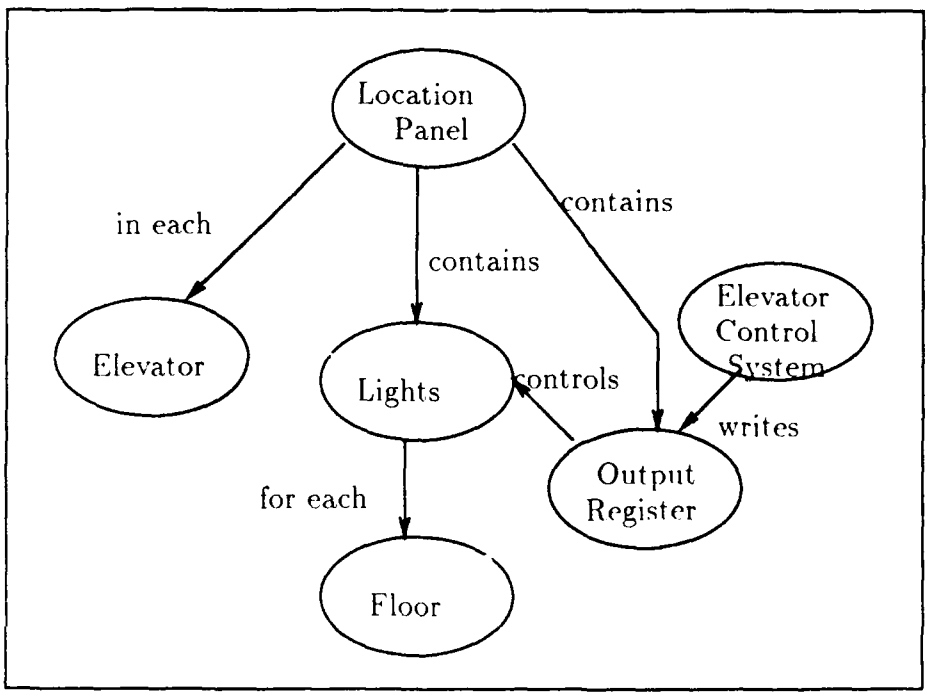


Figure A.7. Elevator Location Panel

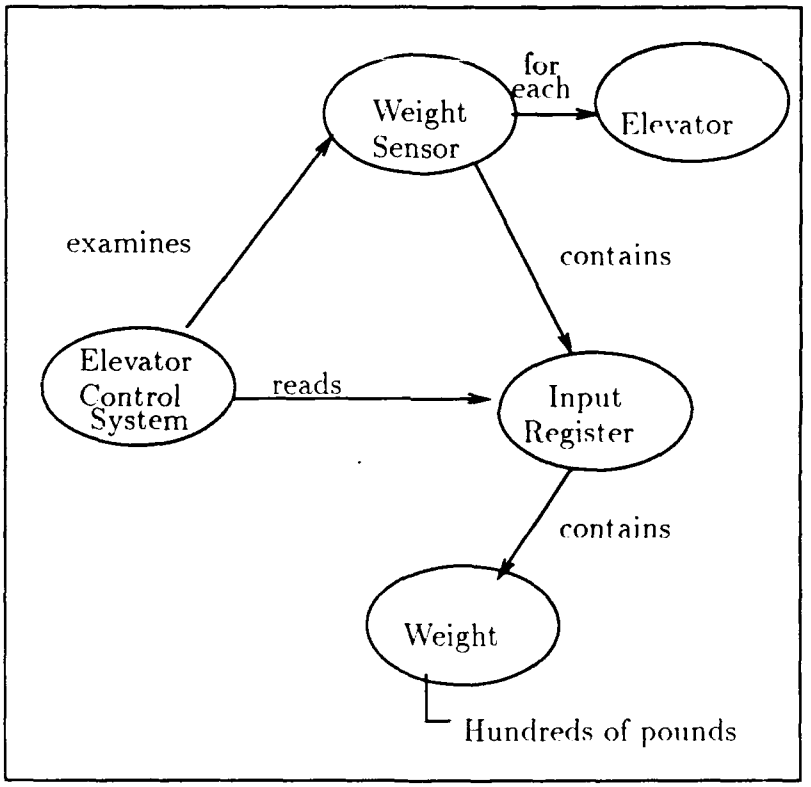


Figure A.8. Elevator Weight Sensor

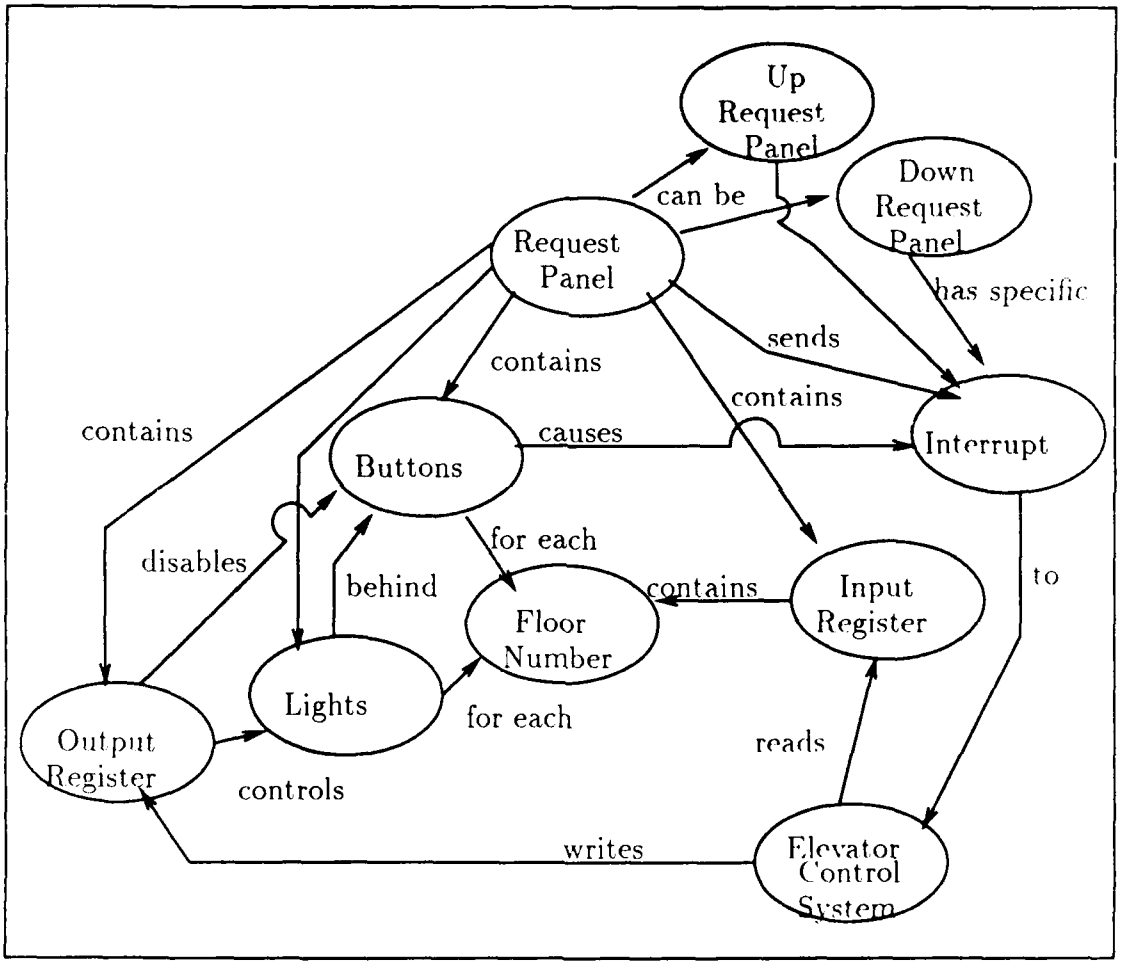
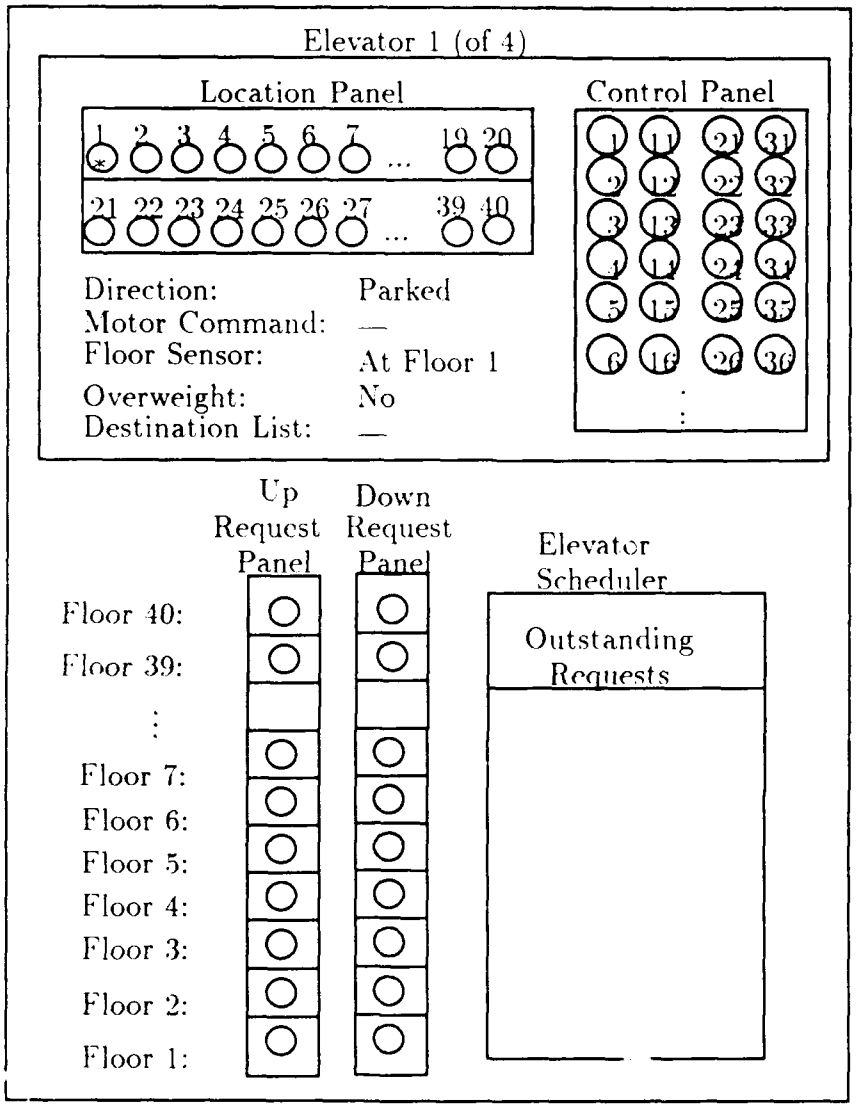


Figure A.9. Elevator Request Panel

A.3 Story Boards

The story boards on the following pages describe some of the situations that the elevator control system will face. The story boards show: 1) the status of one of the four elevators (including the location and control panels, direction of travel, motor commands, etc.); 2) the up and down request panel buttons from each of the floors; and 3) the scheduling algorithm's list of outstanding summons requests.



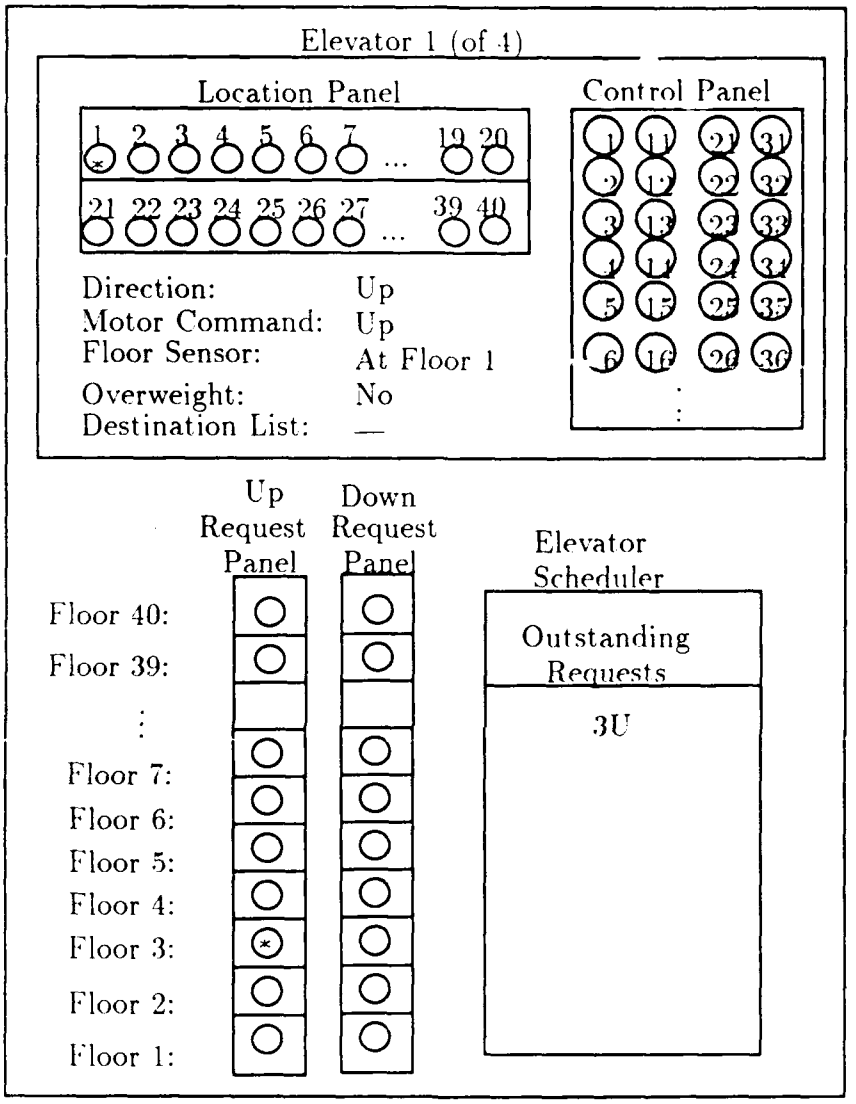
Each of the four elevators has a *location panel* (showing where the elevator currently is) and a *control panel* with buttons for the passengers to enter their destination floor. Lights behind the buttons are lit when the button is pressed. Also associated with an elevator are a direction of travel, commands to the elevator's motor, a sensor signalling which floor the elevator is approaching, an overweight sensor, and a list of destination floors selected by the elevator's passengers.

The *request panels* contain buttons for passengers to summon an elevator. The UP Request Panel contains buttons on all floors except floor 40. Likewise, the DOWN Request Panel contains buttons on all floors except floor 1. The passenger will press the button on either the UP or DOWN Request Panels, depending on his desired direction of travel. The buttons have lights behind them which are lit when the button is pressed.

The *elevator control system* maintains a list of outstanding requests from the request panels, which it uses to schedule elevators to respond to these requests.

An idle elevator is characterized by having "parked" as its direction. In this case, elevator 1 is parked at floor 1, awaiting a command from the elevator control system to respond to a summons request.

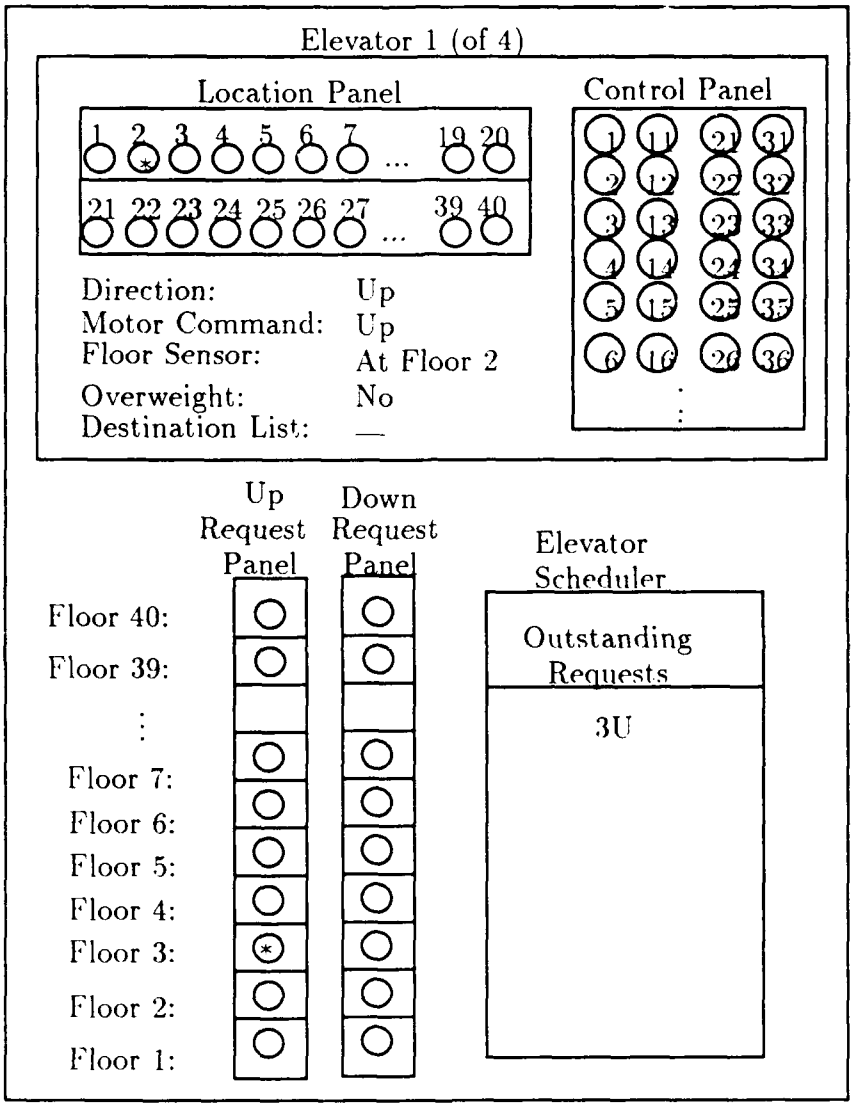
Figure A.10. Story Board: Idle Elevators



The UP Request Panel button on floor 3 is pressed. The elevator control system will:

- illuminate the light behind the "up" button of the request panel on floor 3.
- add the request (3U) to the list of outstanding requests.
- pick an elevator to respond to the request. Since all elevators are currently idle, the closest elevator to the summons (in this case elevator 1) is sent.
- issue an "up" command to elevator 1.
- set elevator 1's direction to "up".

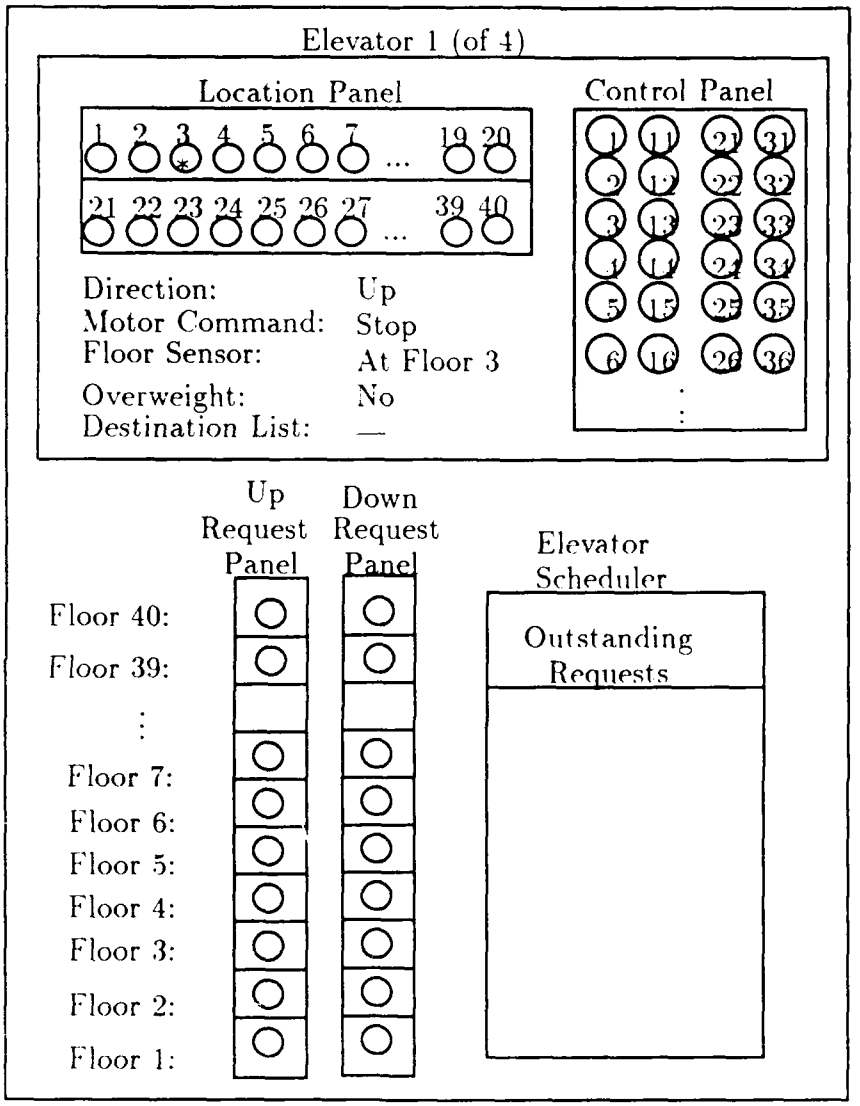
Figure A.11. Story Board: Up Request from Floor 3



The floor sensor for elevator 1 signals arrival of the elevator at floor 2. The elevator control system will:

- extinguish the light for floor 1 on the location panel of elevator 1.
- illuminate the light for floor 2 on the location panel of elevator 1.

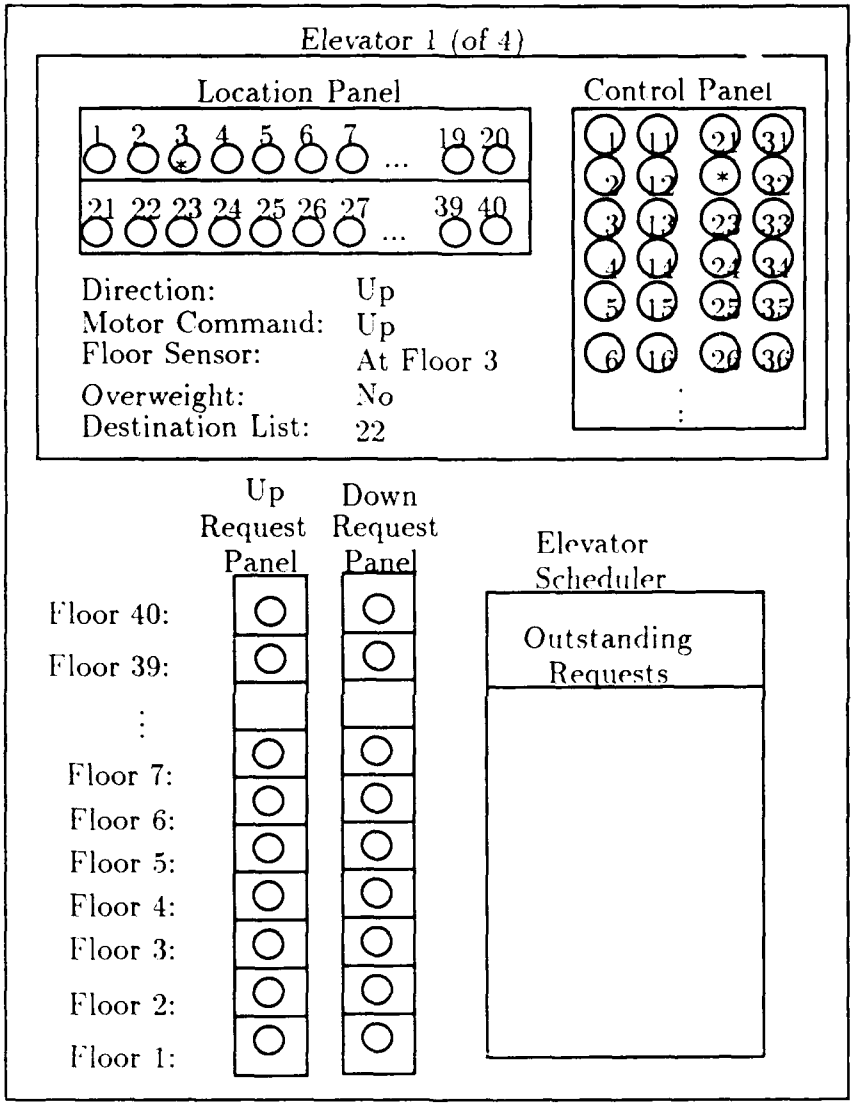
Figure A.12. Story Board: Elevator Arrives at Floor 2



The floor sensor for elevator 1 signals arrival of the elevator at floor 3. The elevator control system will:

- extinguish the light for floor 2 on the location panel of elevator 1.
- illuminate the light for floor 3 on the location panel of elevator 1.
- issue a "stop" command to elevator 1.
- extinguish the light behind the button on floor 3 of the UP Request Panel.
- remove the 3[UP] request from the scheduling algorithm's list of outstanding requests.

Figure A.13. Story Board: Elevator Arrives at Floor 3

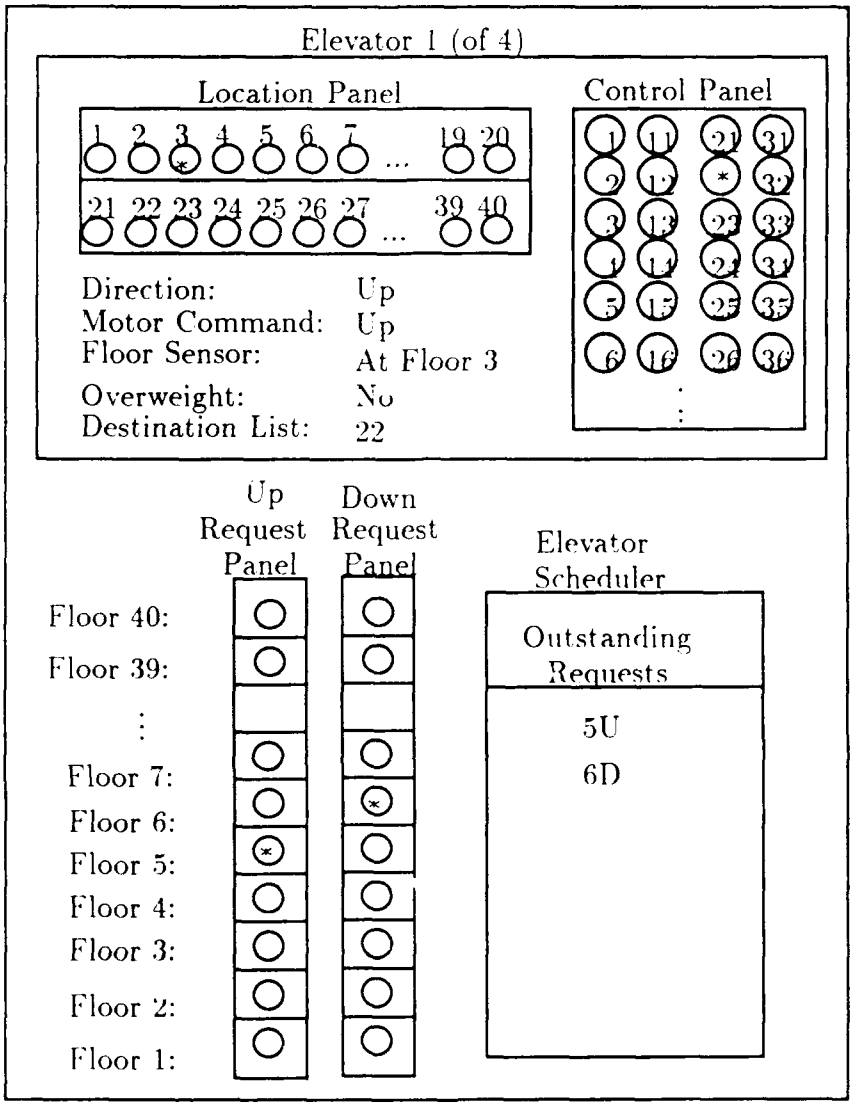


The passenger presses the button for floor 22 on the control panel in elevator 1. The elevator control system will:

- illuminate the light behind button 22 on the control panel for elevator 1.
- add floor 22 to the destination list for elevator 1.
- issue an "up" command to elevator 1.

As elevator 1 rises, its floor sensors will signal the elevator control system to change the lights on elevator 1's location panel accordingly, as in the story board in figure A.3.

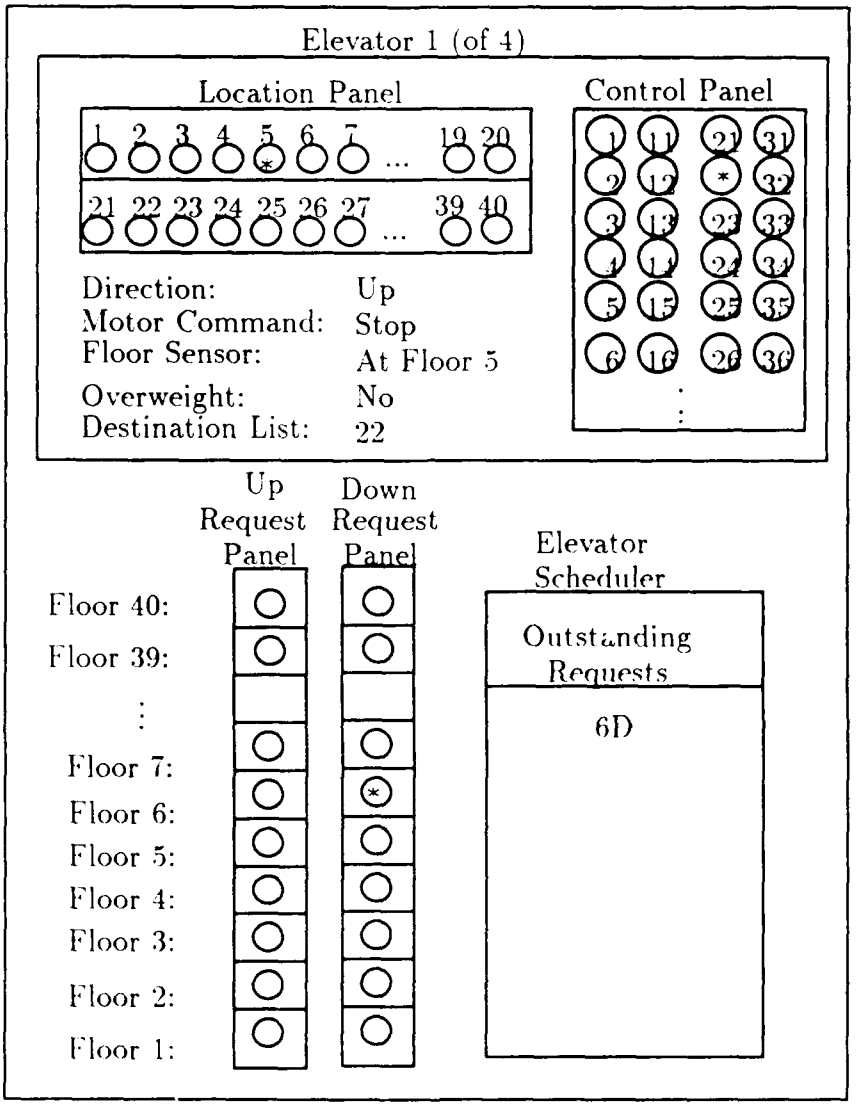
Figure A.14 Story Board: Passenger Presses Destination Button



Additional elevator summons requests are issued from floor 5 (up) and floor 6 (down). The elevator control system will:

- illuminate the light behind the floor 5 button on the UP request panel.
- illuminate the light behind the floor 6 button on the DOWN request panel.
- add the requests (5U & 6D) to the list of outstanding requests.
- pick an elevator to respond to these requests. Since there is currently no elevator heading down, a parked elevator will be sent to respond to the "6D" request. Since elevator 1 is heading up towards floor 5, it may be able to handle the "5U" request. However, the elevator control system may also send a parked elevator to respond to this request since elevator 1 may be delayed for some reason (e.g. overload or emergency stop). The first elevator to reach floor 5 heading up will respond to the request.

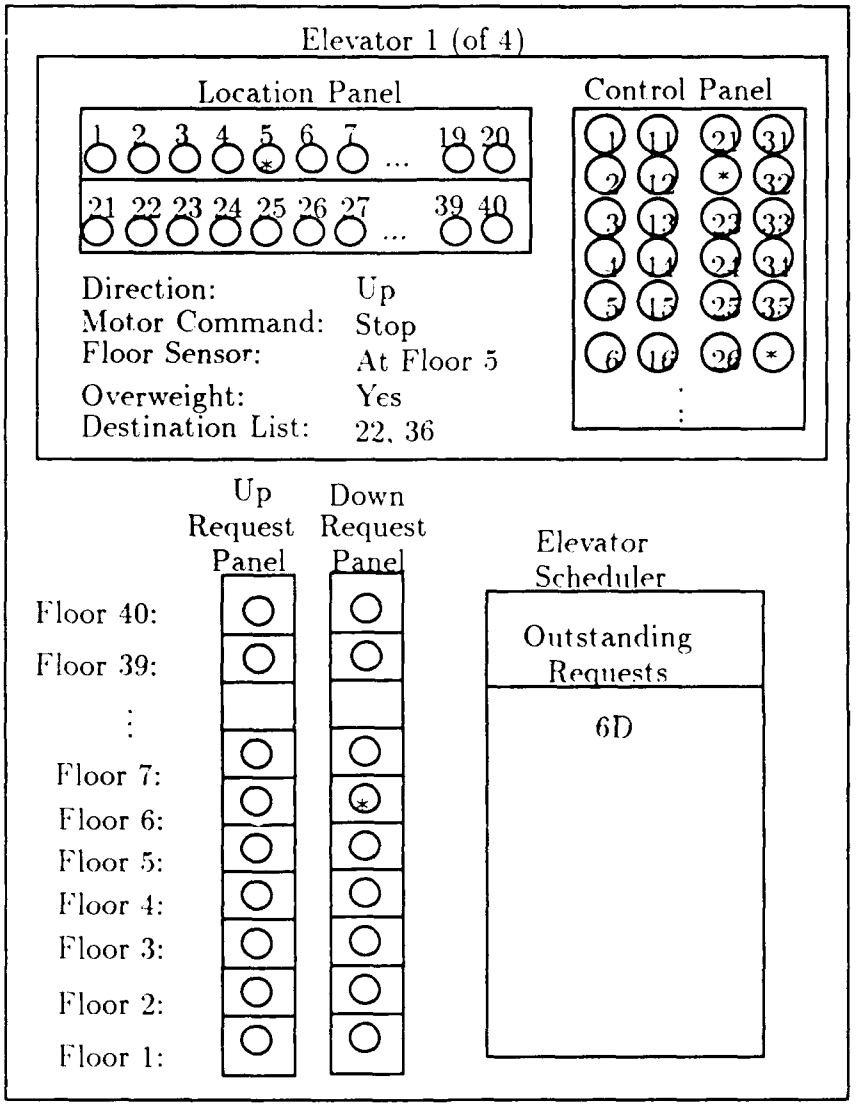
Figure A.15. Story Board: More Summons Requests



The floor sensor for elevator 1 signals arrival of the elevator at floor 5. The elevator control system will:

- extinguish the light for floor 4 on the location panel of elevator 1.
- illuminate the light for floor 5 on the location panel of elevator 1.
- issue a "stop" command to elevator 1.
- extinguish the light behind the floor 5 button of the UP request panel.
- remove the 5U request from the scheduling algorithm's list of outstanding requests.

Figure A.16. Story Board: Elevator Arrives at Floor 3

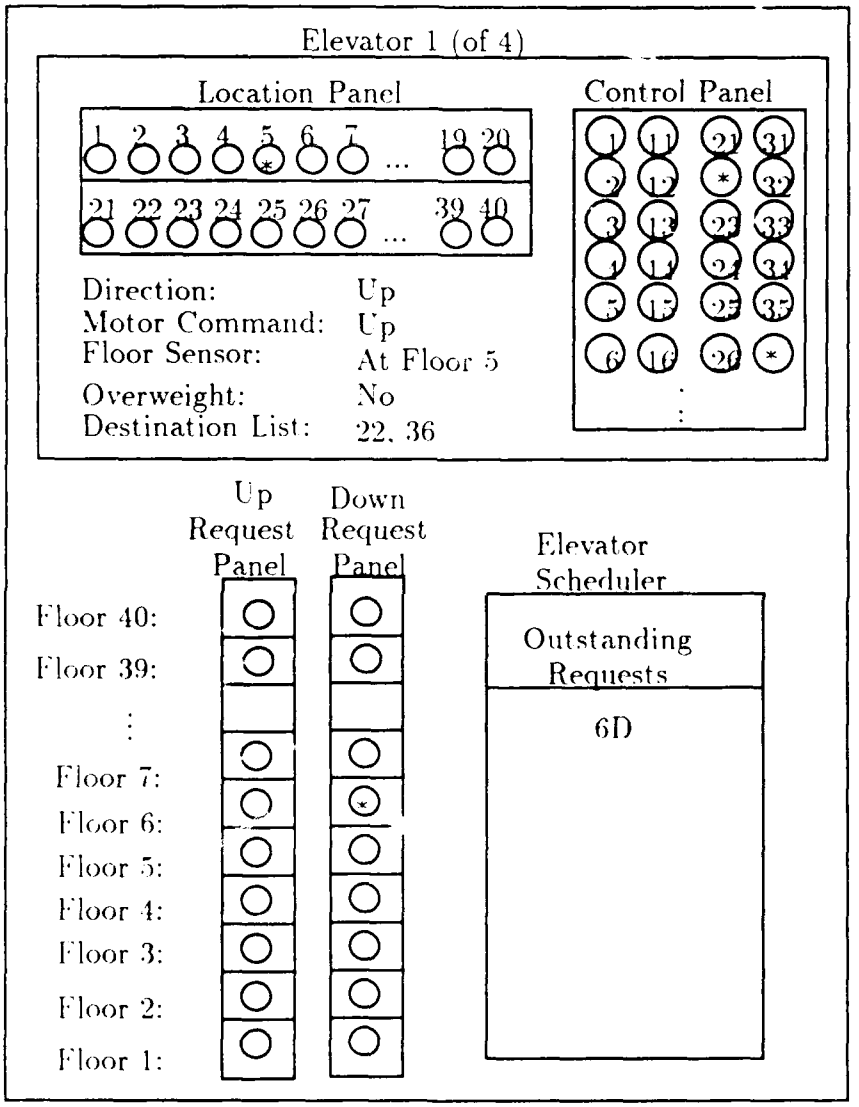


The passenger presses the button for floor 36 on the control panel in elevator 1. However, the elevator is overloaded. The elevator control system will:

- illuminate the light behind button 36 on the control panel for elevator 1.
- add floor 36 to the destination list for elevator 1.
- periodically (approx. every 5 seconds) check the weight sensor and compare with the elevator's maximum capacity.

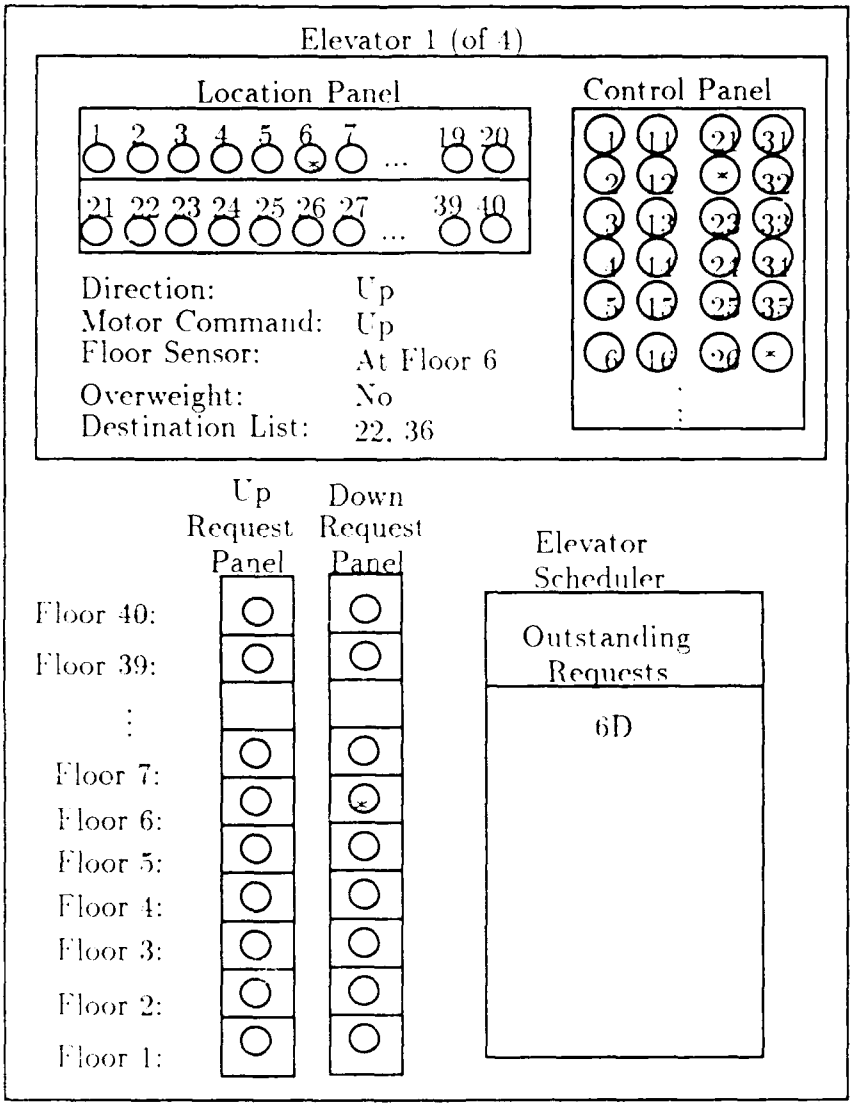
Since the elevator is overloaded, it will not respond to commands from the elevator control system until the overload condition is resolved.

Figure A.17. Story Board: Elevator Overload



Elevator 1 is no longer overloaded, as indicated by the overload sensor. The elevator will now respond to an "up" command from the elevator control system.

Figure A.18. Story Board: Elevator Load Lightened

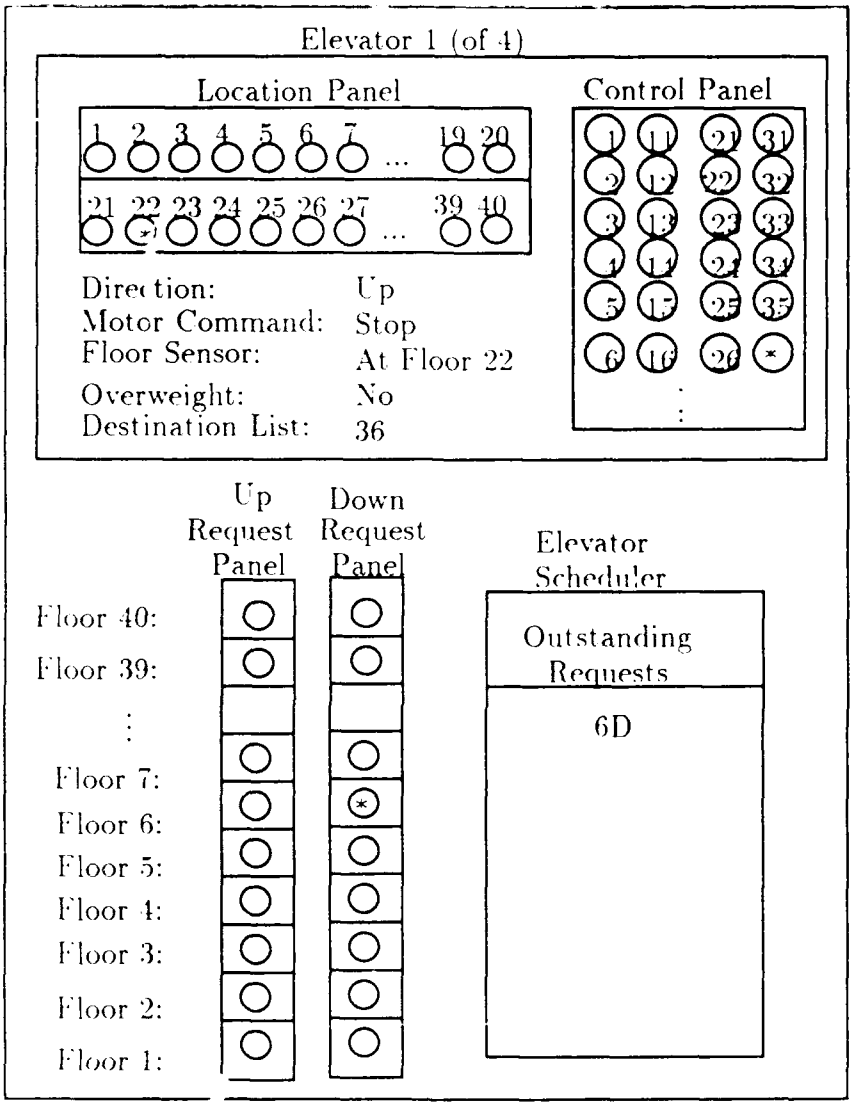


The floor sensor for elevator 1 signals arrival of the elevator at floor 6. The elevator control system will:

- extinguish the light for floor 5 on the location panel of elevator 1.
- illuminate the light for floor 6 on the location panel of elevator 1.

The direction of the summons on floor 6 is "down". Elevator 1 will not stop for a "down" request until it has no [illegible] destinations in the "up" direction. Elevator 1 will continue to rise; its floor sensors will signal the elevator control system to change the lights on elevator 1's location panel accordingly, as in the story board in figure A

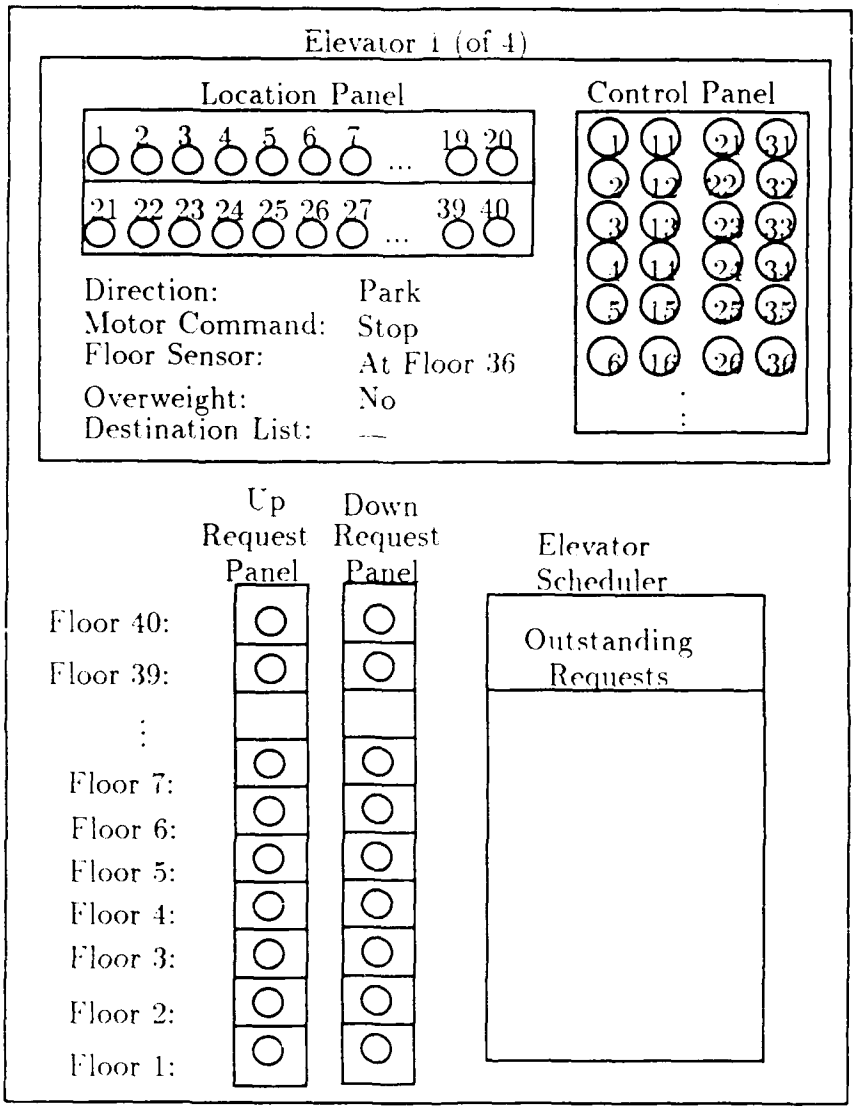
Figure A.19. Story Board: Elevator Passes Floor 6



The floor sensor for elevator 1 signals arrival of the elevator at floor 22. The elevator control system will:

- extinguish the light for floor 21 on the location panel of elevator 1.
- illuminate the light for floor 22 on the location panel of elevator 1.
- issue a "stop" command to elevator 1.
- extinguish the light behind button 22 of the control panel of elevator 1.
- if the destination list for elevator 1 is not empty, wait 3 seconds and issue an "up" command to elevator 1.

Figure A.20. Story Board: Elevator Arrives at Floor 22



The floor sensor for elevator 1 signals arrival of the elevator at floor 36. The elevator control system will:

- extinguish the light for floor 35 on the location panel of elevator 1.
- illuminate the light for floor 36 on the location panel of elevator 1.
- issue a "stop" command to elevator 1.
- extinguish the light behind button 36 of the control panel of elevator 1.

If the destination list for elevator 1 is now empty and there are no outstanding requests (some other elevator is at floor 6) the direction of elevator 1 is set to park.

Figure A.21. Story Board: Elevator Arrives at Floor 36

A.4 *Event/Response List*

Event1: A passenger issues an "up" summons from a particular floor (interrupt).

Resp.1a: Read the Up Summons input register to determine the floor number where the request was made.

R1b: Illuminate the light behind the button on the UP summons request panel.

R1c: If there is an idle (parked) elevator, send it to the floor where the summons was issued.

R1d: Add the request to the list of outstanding requests.

Average response time: The elevator should arrive at the floor in an average of 20 seconds.

E2: A passenger issues a "down" summons from a particular floor (interrupt).

R2a: Read the DOWN Summons *input register* to determine the floor number where the request was made.

R2b: Illuminate the light behind the button on the DOWN summons request panel.

R2c: If there is an idle (parked) elevator, send it to the floor where the summons was issued.

R2d: Add the request to the list of outstanding requests.

Average response time: The elevator should arrive at the floor in an average of 20 seconds.

E3: A sensor for an elevator signals its arrival at a particular floor (interrupt).

R3a: Read the floor number from the floor sensor input register for that elevator.

R3b: Extinguish the light on the location panel for the elevator for the previous floor.

R3c: Illuminate the light on the location panel for the current floor.

R3d: If the floor is listed in the destination list for the elevator, then stop the elevator at the floor and extinguish the light behind the floor number on the elevator's control panel. After stopping, wait 3 seconds, then proceed to the next destination.

R3e: If the floor and direction are listed in the outstanding request list, then stop the elevator at the floor. Extinguish the light behind the floor button on the proper request panel, and remove the summons request from the outstanding request list. After stopping, wait 3 seconds, then proceed to the next destination.

Maximum response time: 0.1 second.

E4: A passenger presses a destination button on the control panel of a particular elevator (interrupt).

R4a: Read the control panel input register to determine the desired floor number.

R4b: Illuminate the light behind the button on the control panel for the elevator.

R4c: Add the floor to the destination list for the elevator.

Maximum response time: 0.1 second.

E5: An elevator becomes overloaded.

R5a: Disable the elevator so that it does not move until the overload condition is gone.

R5b: Periodically (approximately every 5.0 seconds) check to see if the overload is eliminated.

Maximum response time: 0.25 seconds.

E6: Time to check elevator weight sensor (periodic).

R6: If current weight is less than max load, then respond to commands. Otherwise, delay another 5 seconds and check the weight sensor again.

Maximum response time: 2.0 seconds.

A.5 *Known Software Restrictions*

The executable code must fit in 64K of memory. The amount of RAM available for data structures and the program stack is limited to 64K.

The elevator control system should always respond to the passenger pushing a button on the control panel (unless the elevator is overweight, or the emergency stop button is pressed). This should preclude the software from trapping a passenger inside the elevator. The elevator car itself (the hardware) is designed so that the elevator car will only stop at a floor.

A.6 *Metarequirements*

The following paragraphs identify domain expert imposed restrictions on the design of the elevator control system:

The program should schedule the elevators efficiently and reasonably. For example, if someone summons an elevator by pushing the down button on the fourth floor, the next elevator that reaches the fourth floor traveling down should stop at the fourth floor to accept the passenger(s). On the other hand, if an elevator has no passengers (no outstanding destination requests), it should park at the last floor it visited until it is needed again. An elevator should not reverse its direction of travel until its passengers who want to travel in its current direction have reached their destinations.

The maximum weight load for an elevator is 4000 pounds.

An address for a memory mapped input or output register is between 0 and 255. The upper byte is 16#00#.

A floor number is implemented in eight bits, with a range in the current elevator control system of 1..40.

The elevator weight sensors measure the weight of an elevator in hundreds of pounds. The weight value is implemented in eight bits, with a range of 0..255.

Interrupt	Number
Elevator 1 Control Panel	16#01#
Elevator 2 Control Panel	16#02#
Elevator 3 Control Panel	16#03#
Elevator 4 Control Panel	16#04#
Elevator 1 Floor Sensor	16#05#
Elevator 2 Floor Sensor	16#06#
Elevator 3 Floor Sensor	16#07#
Elevator 4 Floor Sensor	16#08#
Up Summons Request Panel	16#0A#
Down Summons Request Panel	16#0B#

Note: The locations given above were not provided in the problem description. However, this information would be available to the analyst.

Table A.1. Elevator Control System Interrupt Numbers

The interrupt numbers used for the hardware signals are shown in table A.1. Addresses of input and output registers are shown in table A.2. Table A.3 shows the control word values for the elevator motor control commands

A.7 *External Interface Diagram*

The external interface diagram for the elevator control system is shown in figure A.7.

Register	Address
Elevator 1 Weight Sensor Register	16#31#
Elevator 2 Weight Sensor Register	16#32#
Elevator 3 Weight Sensor Register	16#33#
Elevator 4 Weight Sensor Register	16#34#
Elevator 1 Control Panel Input Register	16#35#
Elevator 2 Control Panel Input Register	16#36#
Elevator 3 Control Panel Input Register	16#37#
Elevator 4 Control Panel Input Register	16#38#
Elevator 1 Control Panel Output Register	16#39#
Elevator 2 Control Panel Output Register	16#3A#
Elevator 3 Control Panel Output Register	16#3B#
Elevator 4 Control Panel Output Register	16#3C#
Elevator 1 Floor Sensor Input Register	16#41#
Elevator 2 Floor Sensor Input Register	16#42#
Elevator 3 Floor Sensor Input Register	16#43#
Elevator 4 Floor Sensor Input Register	16#44#
Elevator 1 Location Panel Output Register	16#45#
Elevator 2 Location Panel Output Register	16#46#
Elevator 3 Location Panel Output Register	16#47#
Elevator 4 Location Panel Output Register	16#48#
Up Summons Panel Input Register	16#4A#
Down Summons Panel Input Register	16#4B#
Up Summons Panel Output Register	16#4C#
Down Summons Panel Output Register	16#4D#
Elevator 1 Motor Control Register	16#51#
Elevator 2 Motor Control Register	16#52#
Elevator 3 Motor Control Register	16#53#
Elevator 4 Motor Control Register	16#54#

Note: The addresses given above were not provided in the problem description. However, this information would be available to the analyst.

Table A.2. Elevator Control System Register Addresses

Command	Value
Up	16#01#
Down	16#02#
Stop	16#04#

Table A.3. Elevator Motor Control Word Format

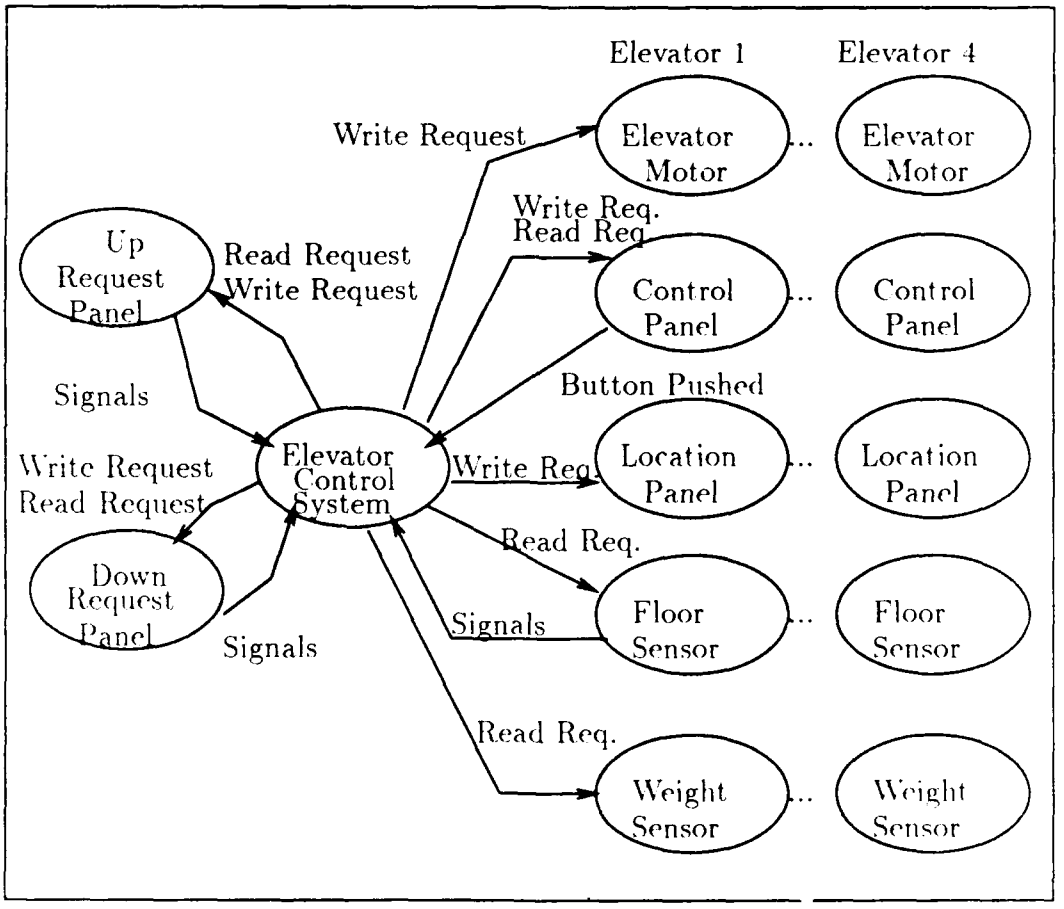


Figure A.22. Elevator Control System External Interface Diagram

A.8 *High Level Actor Object Identification*

The elevator control system is not complex enough to be decomposed into multiple actor objects controlling different problem areas. The elevator control system is documented as a class in the *object encyclopedia.*

A.9 *Organized Preliminary Object List*

```
Elevator Control System

Elevator

    Elevator 1
    Elevator 2
    Elevator 3
    Elevator 4

Direction

    (Associated with each elevator.)

Floor Sensor

    (Associated with each elevator.)

Elevator ID

    (Associated with each elevator.)

Elevator Motor

    (Associated with each elevator.)

Weight Sensor

    (Associated with each elevator.)

Weight
    Current Weight (Associated with each elevator.)
    Load Capacity (Associated with each elevator.)
```

Control Panel

    Elevator Control Panel (Associated with each elevator.)
    UP Request Panel
    DOWN Request Panel

Location Panel

    (Associated with each elevator.)

List

    Destination List (Associated with each elevator.)
    Outstanding Request List

Floor

Summons Request

Input Register

    Elevator Control Panel Input Register (1 for each elevator)
    UP Request Panel Input Register
    DOWN Request Panel Input Register
    Floor Sensor Input Register (1 for each elevator)

Output Register

    Elevator Control Panel Output Register (1 for each elevator)
    UP Request Panel Output Register
    DOWN Request Panel Output Register
    Location Panel Output Register (1 for each elevator)

Address

    (Associated with each input or output register.)

Interrupt Number

    (Associated with each control panel and floor sensor.)

A.10 Message Senders and Receivers

Event1: A passenger issues an "up" summons from a particular floor (interrupt).
Sender: UP Request Panel Receiver: Elevator Control System

Resp.1a: Read the Up Summons input register to determine the floor number of the request. (Performed by UP Request Panel)

R1b: Illuminate the light behind the button on the UP summons request panel. (Performed by UP Request Panel)

R1c: If there is an idle (parked) elevator, send it to the floor where the summons was issued. (Performed by Elevator Control System)

R1d: Add the request to the list of outstanding requests. (Performed by Elevator Control System)

E2: A passenger issues a "down" summons from a particular floor (interrupt).
Sender: DOWN Request Panel
Receiver: Elevator Control System

R2a: Read the DOWN Summons input register to determine the floor number of the request. (Performed by DOWN Request Panel)

R2b: Illuminate the light behind the button on the DOWN summons request panel. (Performed by DOWN Request Panel)

R2c: If there is an idle (parked) elevator, send it to the floor where the summons was issued. (Performed by Elevator Control System)

R2d: Add the request to the list of outstanding requests. (Performed by Elevator Control System)

E3: A sensor for an elevator signals its arrival at a particular floor (interrupt).
Sender: Elevator Floor Sensor
Receiver: Elevator
Forwarded To: Elevator Control System

R3a: Read the floor number from the floor sensor input register for that elevator. (Performed by Floor Sensor)

R3b: Extinguish the light on the location panel for the elevator for the previous floor. (Performed by Elevator)

R3c: Illuminate the light on the location panel for the current floor. (Performed by Elevator)

R3d: If the floor is listed in the destination list for the elevator, then stop the elevator at the floor and extinguish the light behind the floor number on the elevator's control panel. After stopping, remove the floor from the destination list, wait 3 seconds, then proceed to the next destination. (Performed by Elevator)

R3e: If the floor and direction are listed in the outstanding request list, then stop the elevator at the floor. Extinguish the light behind the floor button on the proper request panel, and remove the summons request from the outstanding request list. After stopping, wait 3 seconds, then proceed to the next destination. (Performed by Elevator Control System)

E4: A passenger presses a destination button on the control panel of a particular elevator (interrupt).
Sender: Elevator Control Panel
Receiver: Elevator

R4a: Read the control panel input register to determine the desired floor number. (Performed by Control Panel)

R4b: Illuminate the light behind the button on the control panel for the elevator. (Performed by Control Panel)

R4c: Add the floor to the destination list for the elevator. (Performed by Elevator)

E5: An elevator becomes overloaded.
Inquiry Sender: Elevator
Inquiry Receiver: Weight Sensor

R5a: Disable the elevator so that it does not move until the overload condition is gone. (Performed by Elevator)

R5b: Periodically (approximately every 5.0 seconds) check to see if the overload is eliminated. (Performed by Elevator)

E6: Time to check elevator weight sensor (periodic).

R6: If current weight is less than max load, then respond to commands. Otherwise, delay another 5 seconds and check the weight sensor again. (Performed by Elevator)

A.11 *Documentation of Object Classes*

The remaining pages of this appendix document the object classes as entries in the *Object Encyclopedia.*

Elevator Control System

Textual Description:

The Elevator Control System is the main object of the system. It contains software models of the UP and DOWN request panels, and each of the four elevators. This is a high level actor object which keeps track of and schedules each of the elevators.

The elevator control system keeps track of the outstanding summons requests and schedules elevators to meet these requests. The elevator control system should schedule the elevators efficiently and reasonably. For example, if someone summons an elevator by pushing the down button on the fourth floor, the next elevator that reaches the fourth floor traveling down should stop at the fourth floor to accept the passengers. When the elevator control system receives a summons request, it should *first determine if there is a parked elevator to send to answer the summons.* If not, when an elevator reaches its final destination and the request is still outstanding, the free elevator should be sent to the floor to answer the summons.

Structure Diagram, Interface Diagram, and State Transition Diagram:

See figures A.11, A.24, and A.25.

Messages received by class:

Arrived at Floor	A signal that an elevator has arrived at a floor.
Up Summons Request	A signal than an UP summons request has been issued.
Down Summons Request	A signal than a DOWN summons request has been issued.

Messages sent by class:

Summons Request.Create	Create a summons request from the floor number and direction .
Control Panel.Extinguish Light	Signal the UP or DOWN request panel to extinguish one of its lights due to an elevator arrival.
List.Is Empty	Test if the list is empty.
List.Add Item	Add a summons request to the list.
List.Remove Item	Remove a summons request from the list.
List.Is In List	Checks to see if a given summons request is in the list.
Floor.Assignment	Assign one value of a floor number to another.
Floor.Is less than	Test if one floor number is less than the other.
Elevator.Final Destination Of	Returns the final destination of the elevator.
Elevator.Direction Of	Returns the direction of the elevator.
Elevator.Floor Number Of	Returns the current floor where the elevator is.
Elevator.Set Direction	Set the direction of an idle elevator.
Elevator.Stop at this Floor	Signal to the elevator to stop at the current floor.
Elevator.Go To Floor	Signal to the elevator to go the the given floor.

Elevator ID.Assignment	Assign one value of an Elevator ID to another.
Elevator ID.Is equal	Test if two elevator IDs are equal.

Description of any state limitations:

If an outstanding request is still pending after 20 seconds, poll the elevators to see if any is free to respond to the summons.

List of exported exceptions:

None. The Elevator Control System must handle all errors internally.

List of exported constants:

None.

Re-use considerations:

This class is application specific.

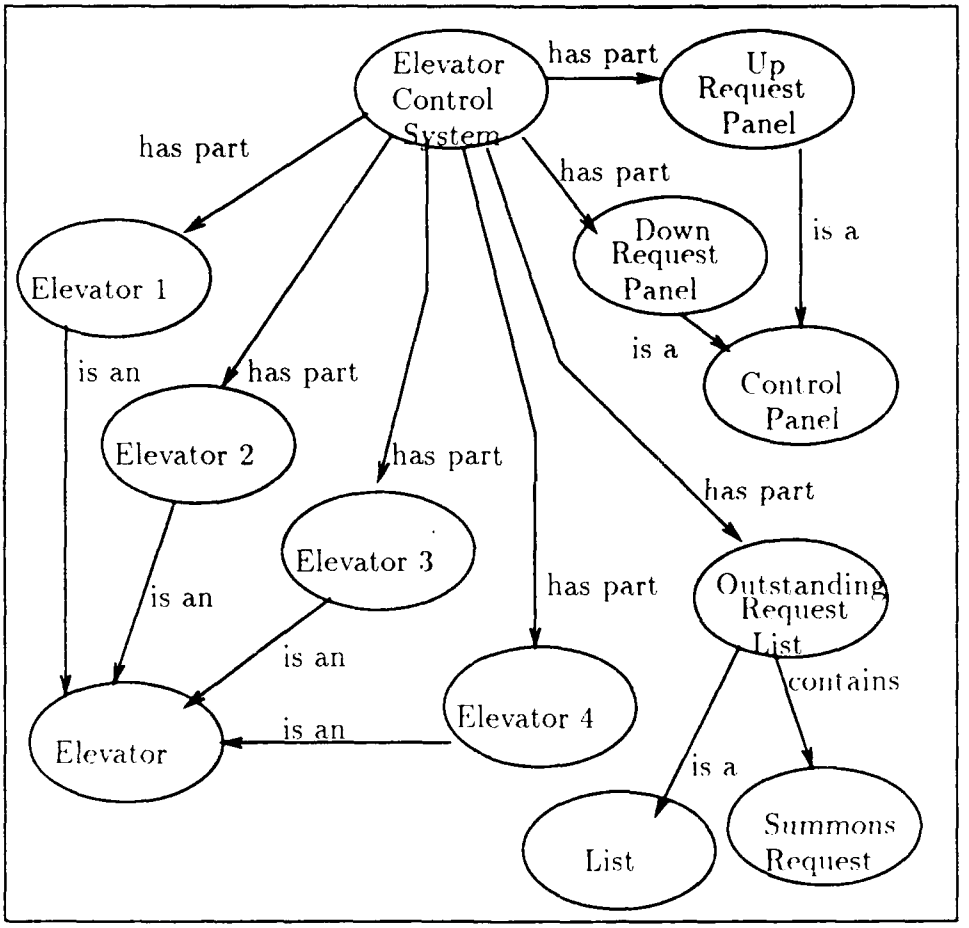


Figure A.23. Elevator Control System: Structure Diagram

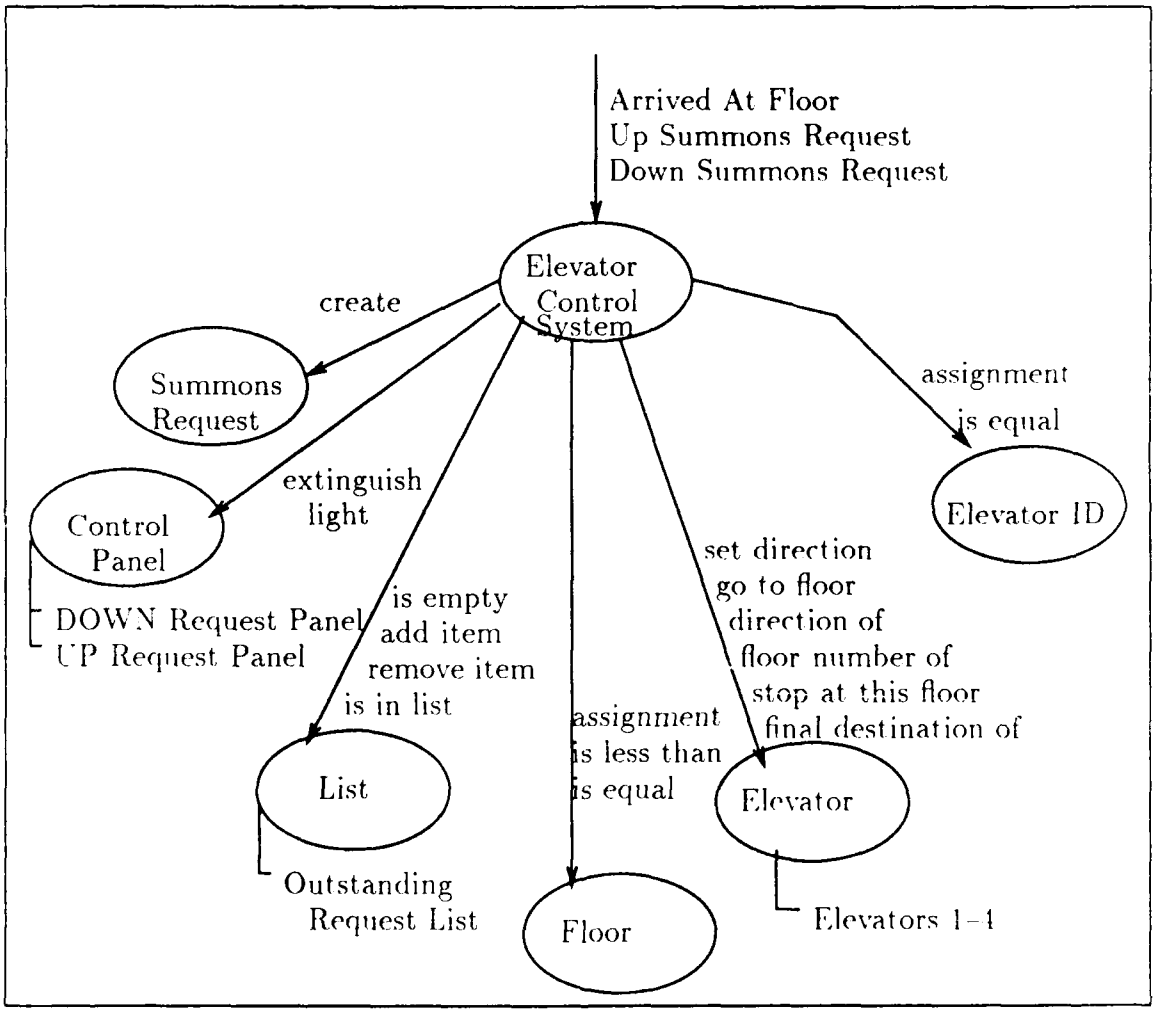


Figure A.24. Elevator Control System: Interface Diagram

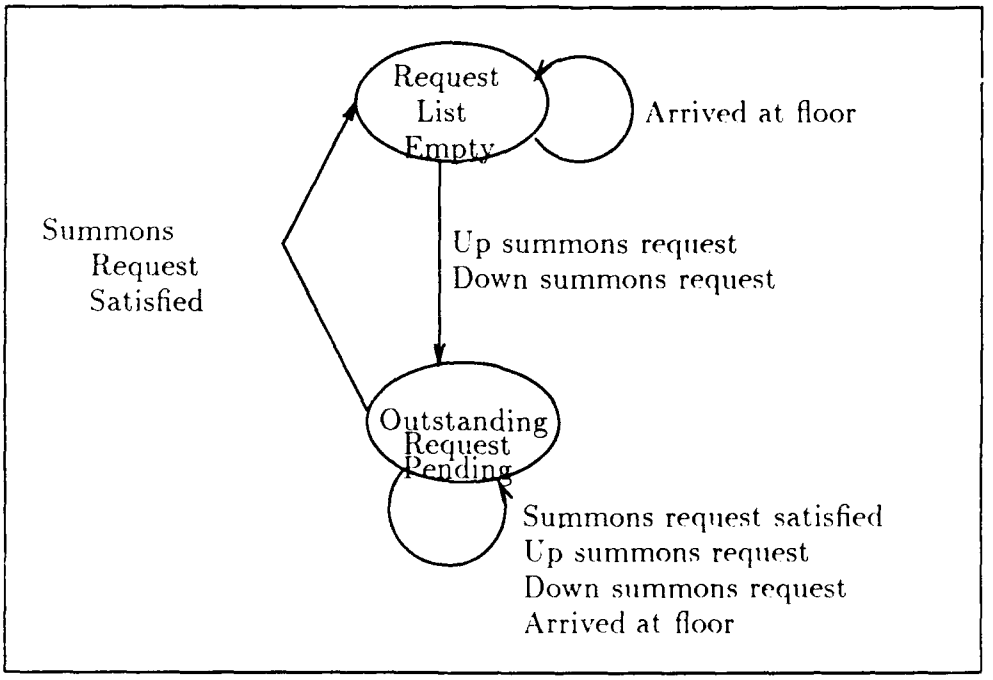


Figure A.25. Elevator Control System: State Transition Diagram

Elevator

Textual Description: An elevator object controls the movement of a single elevator. This class of objects, each of which has a unique Elevator ID, contains controllers for each logical component of the elevator: motor, control panel, location panel, weight sensor, and floor sensor. The elevator has a direction associated with it, and maintains a list of destinations entered on its control panel. This controller will move the elevator to each floor in its destination list, and respond to requests relayed through the elevator control system. The controller handles the setting of lights on the location panel based on input from the floor sensors. An elevator with no outstanding destination requests should park at the last floor it visited. An elevator should not reverse its direction of travel until it has reached its final destination in its current direction.

Structure Diagram, Interface Diagram, and State Transition Diagram:

See figures A.26, A.27, and A.28.

Messages received by class:

Go To Floor	Stop the elevator at the given floor.
Stop At This Floor	Stop the elevator at the current floor.
Set Direction	Sets the direction of the elevator. (The elevator direction must first be "Parked".)
Floor Number Of	Returns the current location (floor number) where the elevator is.

Direction Of	Returns the current direction of the elevator.
Final Destination Of	Returns the floor number of the elevators final destination in its destination list.
Button Pushed	This is a signal that a given button has been pushed on the control panel, indicating a new destination.
Arrival At Floor	This is a signal from the elevator floor sensor that the elevator is about to arrive at a given floor.

Messages sent by class:

Elevator Motor.Up	Signal the motor to move the elevator up.
Elevator Motor.Down	Signal the motor to move the elevator Down.
Elevator Motor.Up	Signal the motor to stop the elevator.
Control Panel.Extinguish Light	Signal the elevator control panel to extinguish the light for the current floor.
Elevator ID.Assignment	Assign one Elevator ID to another.
Location Panel.Illuminate Light	Signal the elevator location panel to illuminate the light of the floor at which the elevator is about to arrive.
Location Panel.Extinguish Light	Signal the elevator location panel to extinguish the light of the floor the elevator is leaving.

Elevator Control System.Arrived At Floor	Signal to the elevator control system the elevator has just arrived at a floor.
Weight Sensor.Check Weight	Check the current weight of the elevator.
Weight.Assignment	Assign the weight sensor value to the current weight attribute.
Weight.Is Less Than	Determine if the current weight is less than the elevator's load capacity.
Direction.Assignment	Set the value of the elevator's direction attribute.
Direction.Is Equal To	Test the current value of the direction attribute to see if it is "Parked".
List.Is Empty	Check if the destination list is empty.
List.Is In List	Check if a floor number is in the destination list.
List.Remove Item	Remove a floor from the destination list when you arrive at that floor.
List.Add Item	Add a floor to the destination list.

Description of any state limitations:

If the elevator is overweight, the elevator controller will periodically (every five seconds) check the weight sensor to see if the load has been reduced. The elevator will not leave the "overweight" state until the sensor reports that the value of current weight is less than the load capacity.

List of **exported exceptions:**

Elevator Busy	An attempt was made to set the direction of an elevator whose direction was not "Parked".
Elevator Overweight	An attempt was made to direct the elevator to another floor when the elevator is overweight.

List of **exported constants:**

None.

List of **objects in class:**

- Elevator 1
- Elevator 2
- Elevator 3
- Elevator 4

Re-use considerations:

This class is application specific.

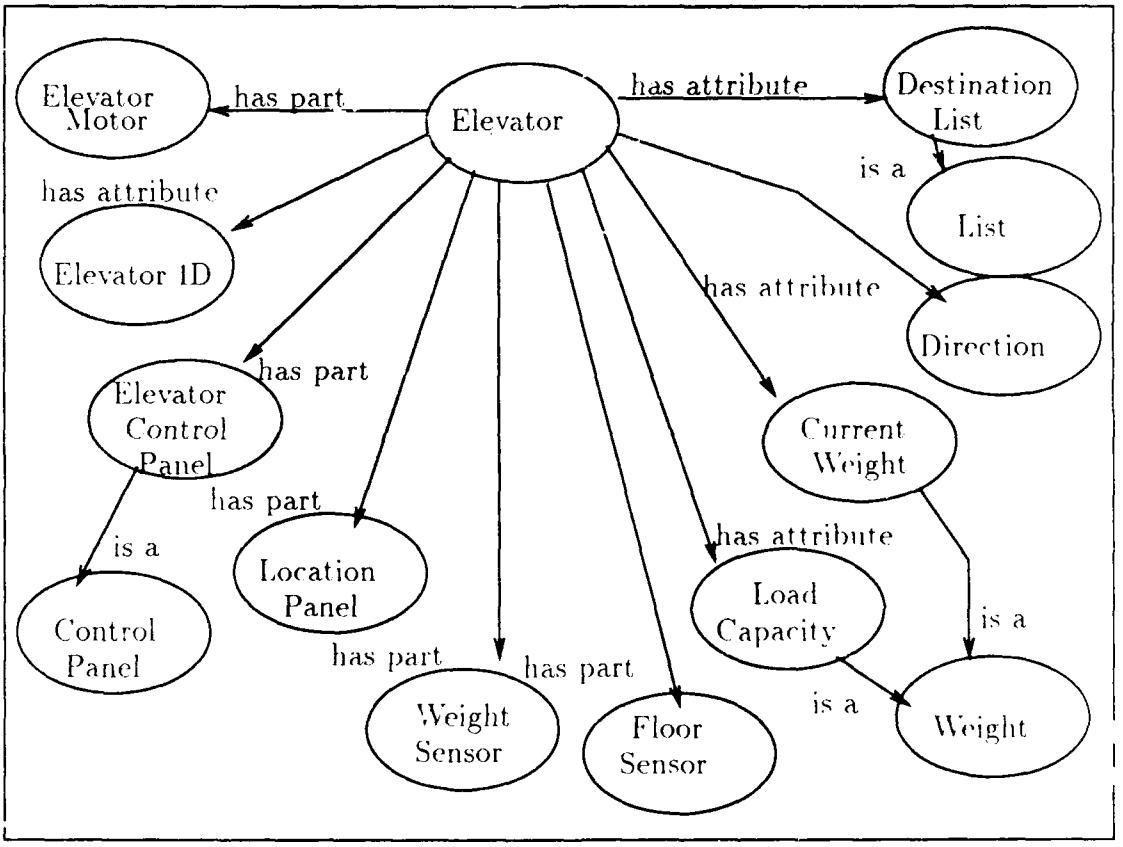


Figure A.26. Elevator: Structure Diagram

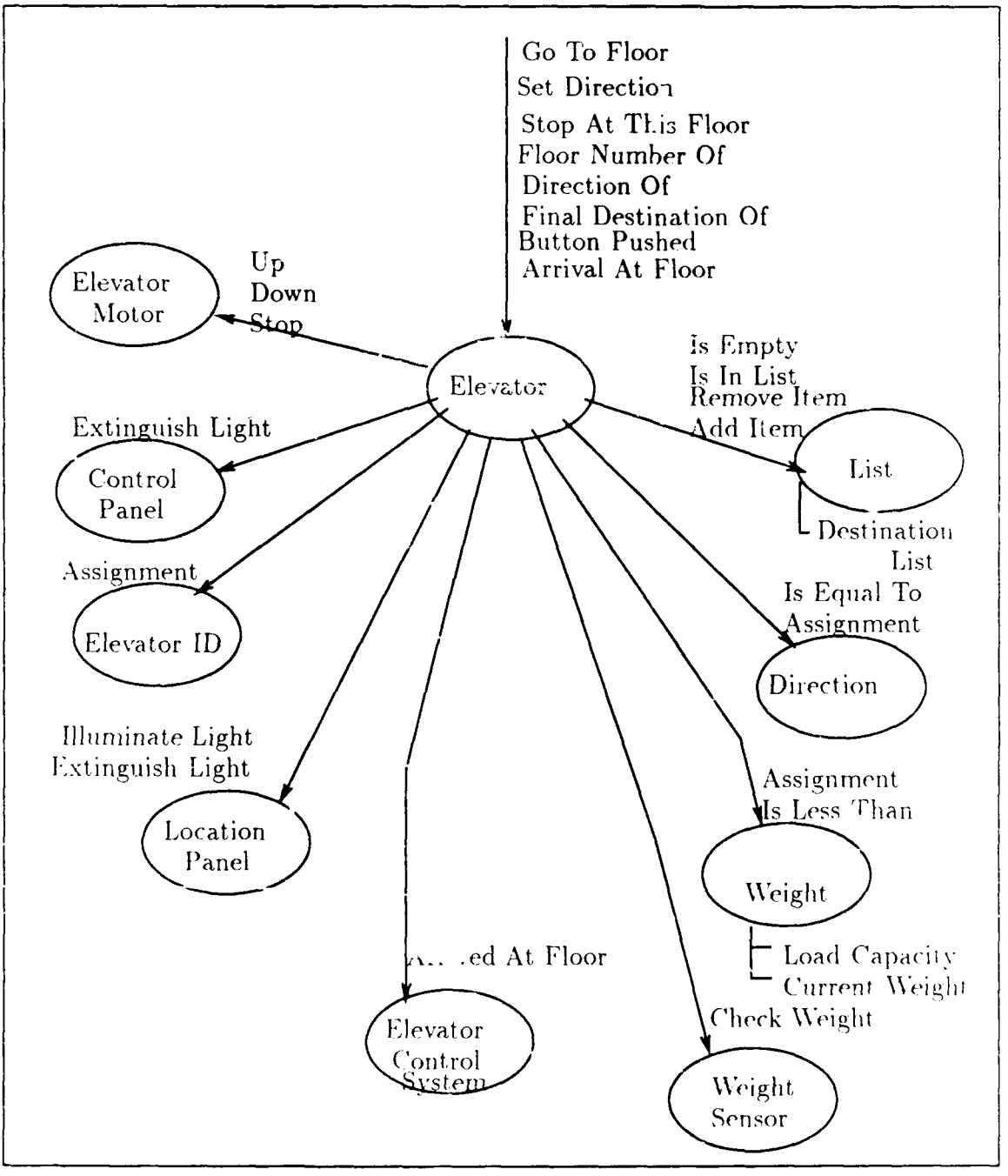


Figure A.27. Elevator: Interface Diagram

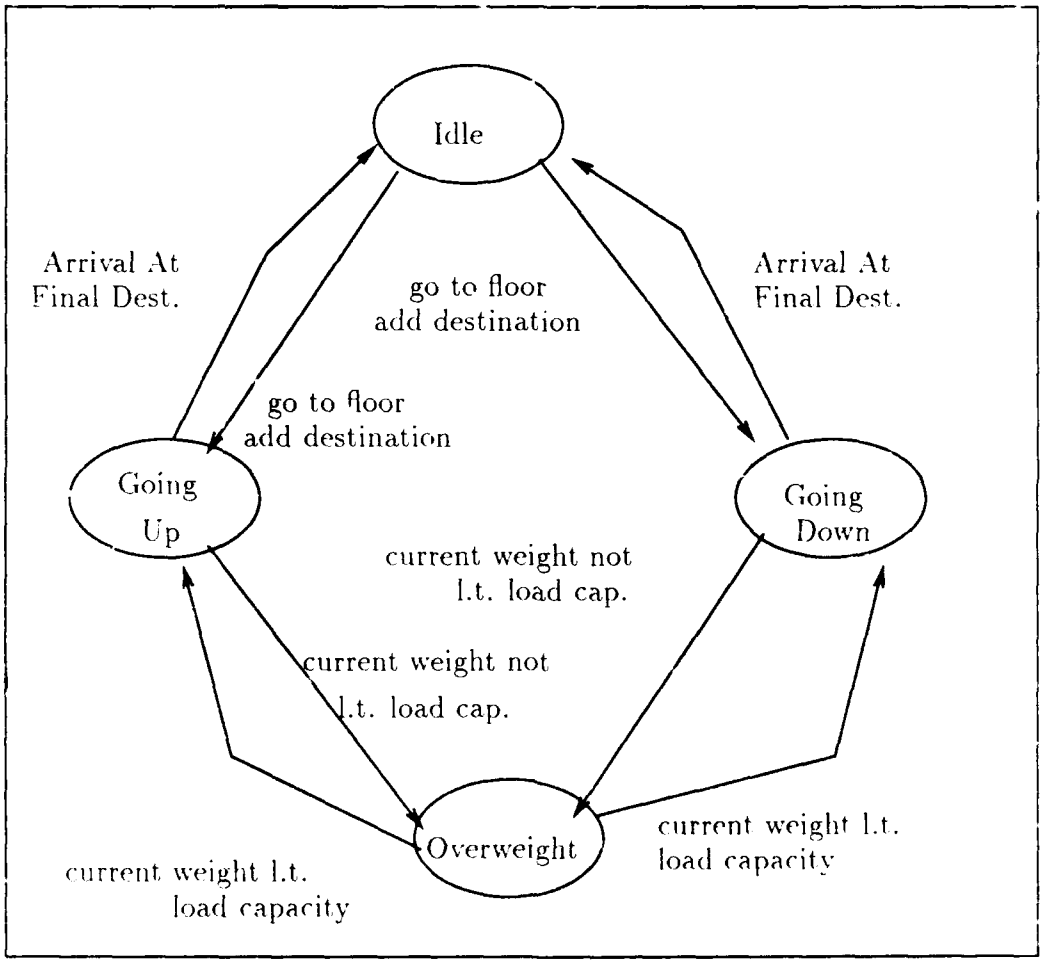


Figure A.28. Elevator: State Transition Diagram

Control Panel

Textual Description:

An object of the control panel class drives a hardware control panel. It handles the interrupts raised by pressing buttons on the panel, determines which button is pressed, and automatically lights the lamps behind the buttons. The control panel is contained within a parent object, and sends a message to this object when a button is pushed. The control panel uses input and output registers to communicate with the hardware.

Structure Diagram, Interface Diagram, and State Transition Diagram:

See figures A.29, A.30, and A.31.

Messages received by class:

Extinguish Light	Extinguish a light behind a button on the control panel.
Button Pushed	This is a signal (interrupt) from the hardware that a button on the control panel has been pushed and the button number is in the input register.

Messages sent by class:

Interrupt Number.Assignment	Assign a value of one object to another when initializing the control panel.
Address.Assignment	Assign a value of one address object (for the input and output registers) to another when initializing the control panel.
Button Pushed	A message to the receiver that a control panel button has been pushed.

Input Register.Read	Read the floor number from the input register.
Output Register.Write	Write the floor number to the output register.
Floor.Assignment	Assign the value of one object to another floor object.

Description of any state limitations:

When the control panel writes a floor number to the output register, that light is toggled. Therefore, the message *Extinguish Light* could actually illuminate the light. The control panel driver assumes that the light specified in this message is actually on. This is a safe assumption in this system since the elevator or elevator control system will maintain a list (destination list and outstanding response list) of those control panel buttons that are pressed.

List of exported exceptions:

None.

List of exported constants:

None.

List of objects in class:

- Elevator 1 Control Panel
- Elevator 2 Control Panel
- Elevator 3 Control Panel
- Elevator 4 Control Panel
- UP Request Panel
- DOWN Request Panel

Re-use considerations:

This class has re-use potential. If re-used, take note of the discussion under the *State Limitations* section.

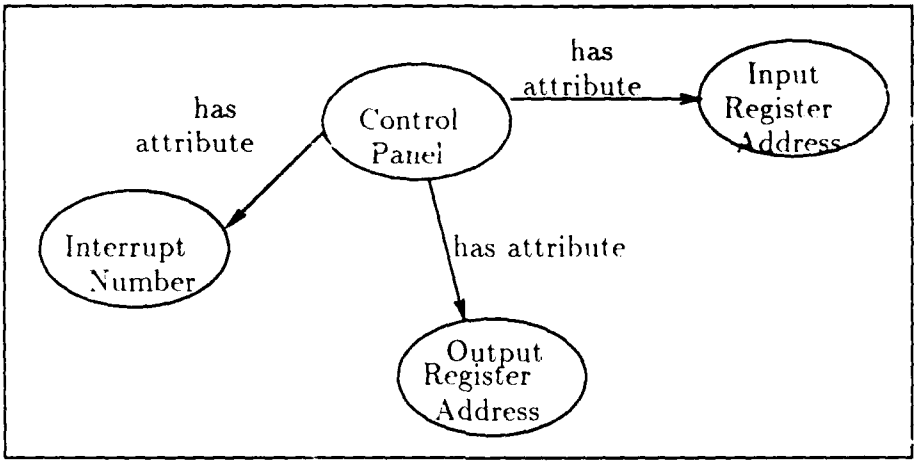


Figure A.29. Control Panel: Structure Diagram

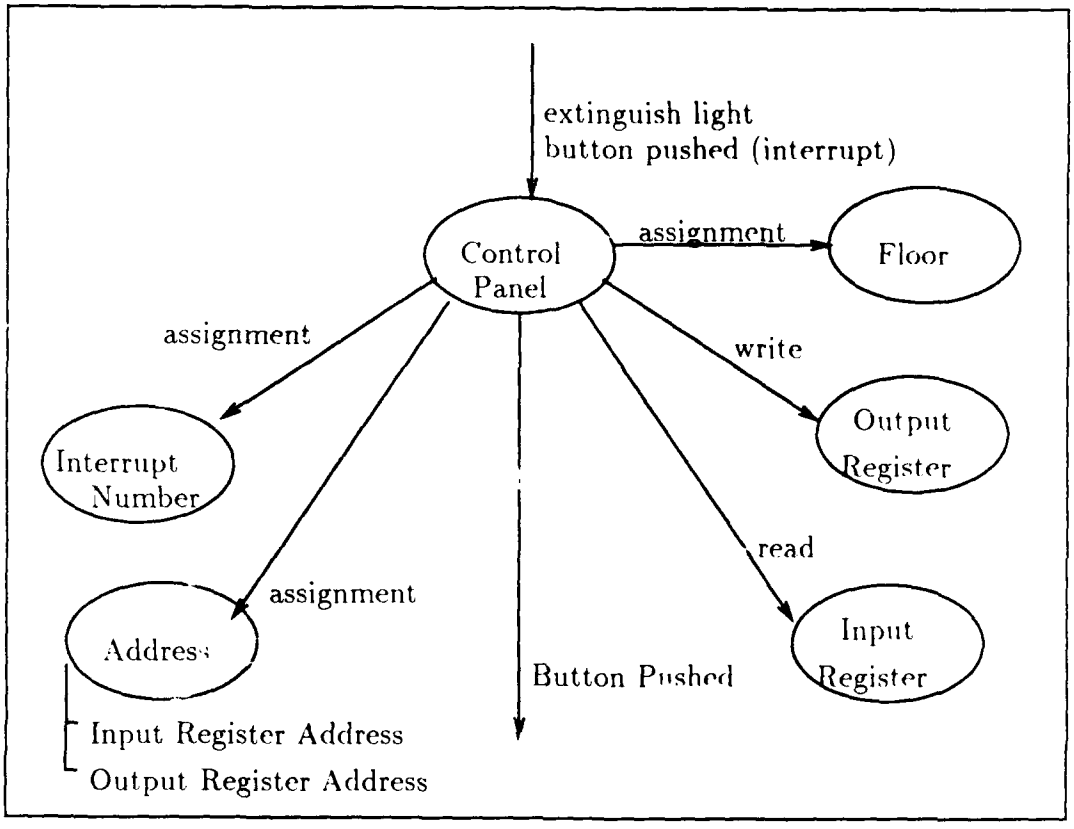


Figure A.30. Control Panel: Interface Diagram

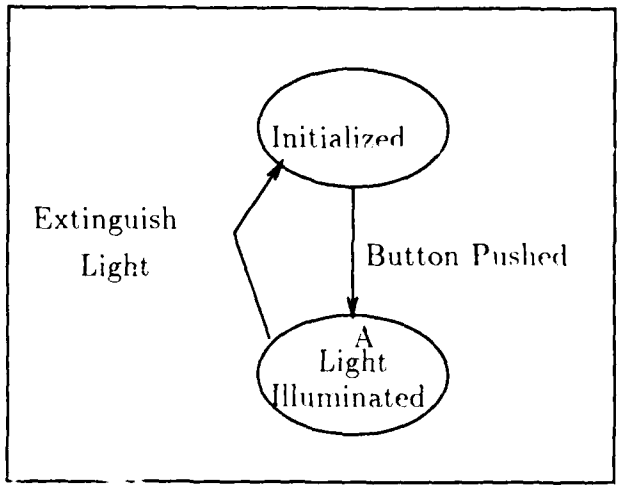


Figure A.31. Control Panel: State Transition Diagram

Address

Textual Description:

An address specifies the memory location of an input or output register. The address is implemented in eight bits.

Structure Diagram, Interface Diagram, and State Transition Diagram:

See figures A.32, A.33, and A.34.

Messages received by class:

Assignment	Assign the value of one address object to another.

Messages sent by class:

None.

Description of any state limitations:

An object of type address must have a value in the range 0..256. An address object must be initialize before it can be read.

List of exported exceptions:

Constraint Error	The value assigned to the object is not in the proper range

List of exported constants:

Max Address — 256

Re-use considerations:

This class is potentially re-usable.

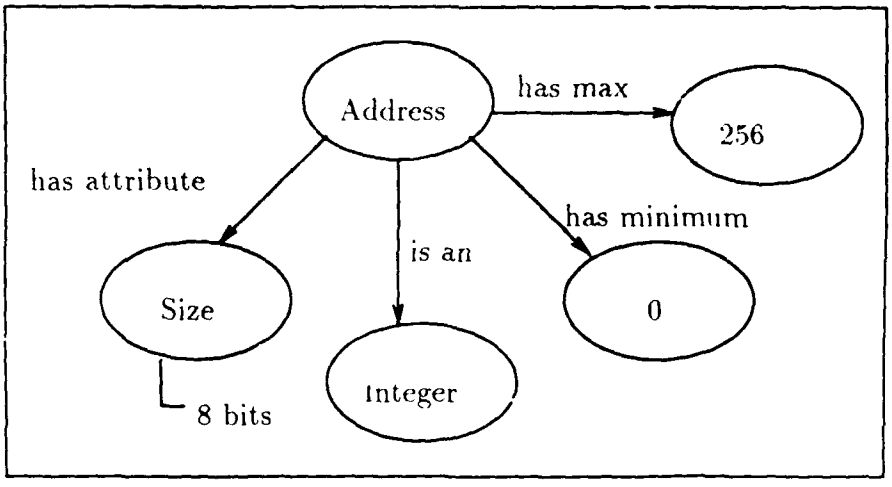


Figure A.32. Address: Structure Diagram

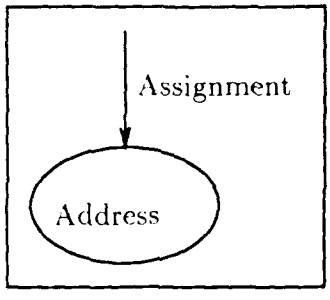


Figure A.33. Address: Interface Diagram

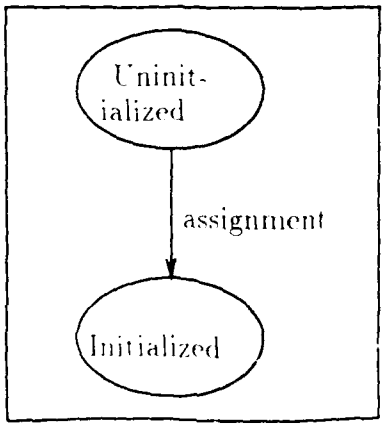


Figure A.34. Address: State Transition Diagram

Direction

Textual Description:

The direction class specifies the direction of an elevator. The values of an object of this class are "UP", "DOWN", or "Parked".

Structure Diagram, Interface Diagram, and State Transition Diagram:

See figures A.35, A.36, and A.37.

Messages received by class:

Assignment	Assign the value of one direction object to another.
Is Equal To	Test to see if two direction objects have the same value.

Messages sent by class:

None.

Description of any state limitations:

The values of an object of this class are "UP", "DOWN", or "Parked". An object of this class may not be read until it has been assigned a value.

List of exported exceptions:

Constraint Error	The value assigned to an object of this class is not "UP", "DOWN", or "Parked".

List of exported constants:

The following values are visible: "UP", "DOWN", or "Parked".

Re-use considerations:

This class is application specific.

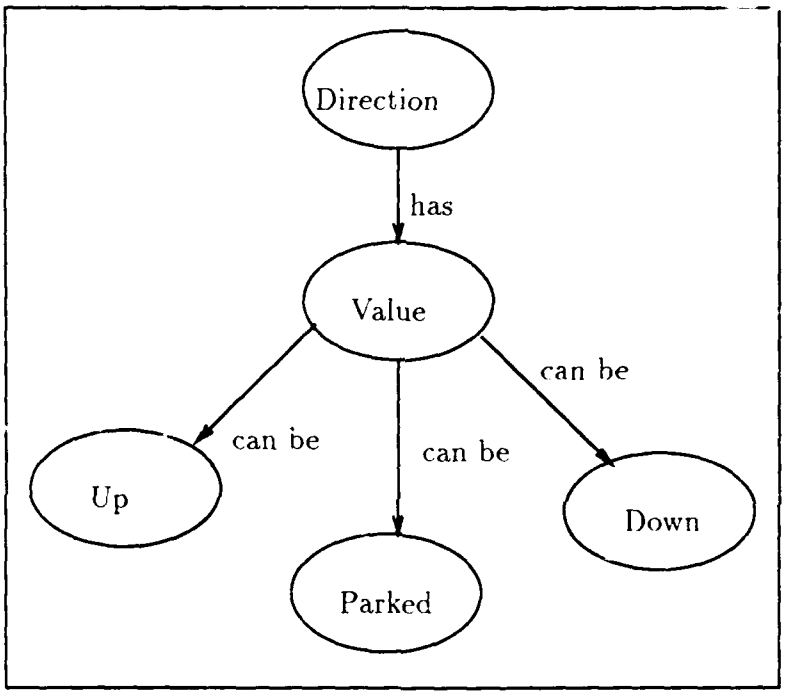


Figure A.35. Direction: Structure Diagram

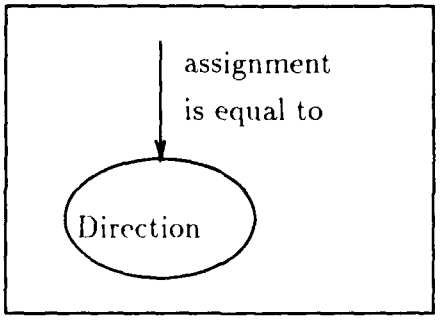


Figure A.36. Direction: Interface Diagram

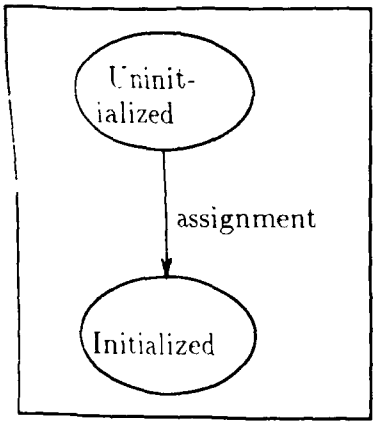


Figure A.37. Direction: State Transition Diagram

Elevator ID

Textual Description:

An elevator ID is simply an integer identifier used to identify an elevator object.

Structure Diagram, Interface Diagram, and State Transition Diagram:

See figures A.38, A.39, and A.40.

Messages received by class:

Assignment	Assign the value of one Elevator ID object to another.
Is equal	Test if two Elevator IDs are equal.

Messages sent by class:

None.

Description of any state limitations:

The value of an object of this class is limited to the range 1..4. An object of this class may not be read until in has been assigned a value.

List of exported exceptions:

Constraint Error	The value assigned to an object of this class is not in the range 1..4.

List of exported constants:

None.

Re-use considerations:

This class is application specific.

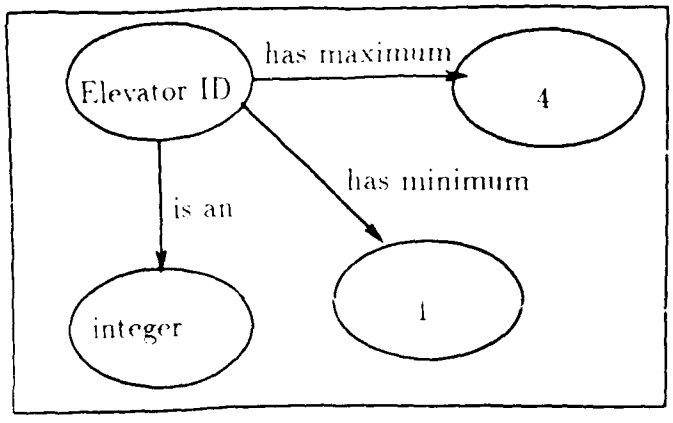


Figure A.38. Elevator ID: Structure Diagram

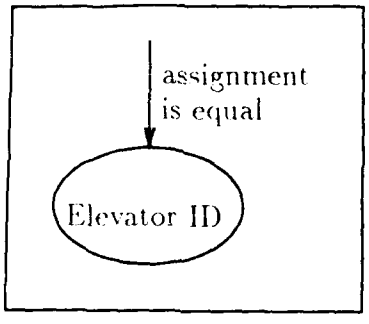


Figure A.39. Elevator ID: Interface Diagram

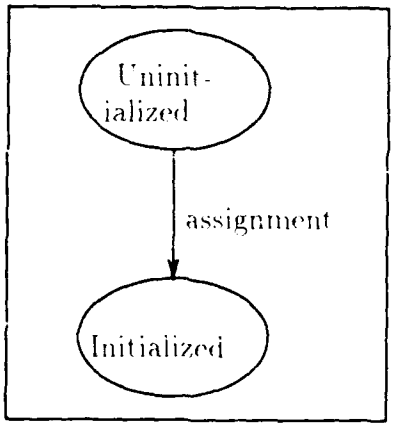


Figure A.40. Elevator ID: State Transition Diagram

Elevator Motor

Textual Description:

This class of objects controls the motor of an elevator. It responds to commands sent as messages by loading the proper control word into the physical output port that controls the elevator motor. The commands are: Up — 16#01#, Down 16#02#, Stop - 16#04#.

Structure Diagram, Interface Diagram, and State Transition Diagram:

See figures A.41, A.42, and A.43.

Messages received by class:

Up	Set the motor to raise the elevator.
Down	Set the motor to lower the elevator.
Stop	Set the motor to stop the elevator.

Messages sent by class:

Output Register.Write	Write a control word to the output register.
Address.Assignment	Assign one address object's value to another.

Description of any state limitations:

None.

List of exported exceptions:

None.

List of exported constants:

None.

List of objects in class:

- Elevator 1 Motor
- Elevator 2 Motor
- Elevator 3 Motor
- Elevator 4 Motor

Re-use considerations:

This class is application specific.

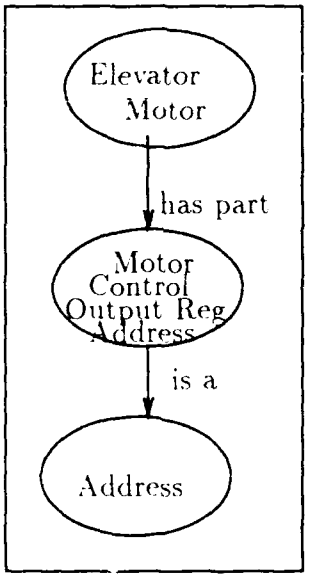


Figure A.41. Elevator Motor: Structure Diagram

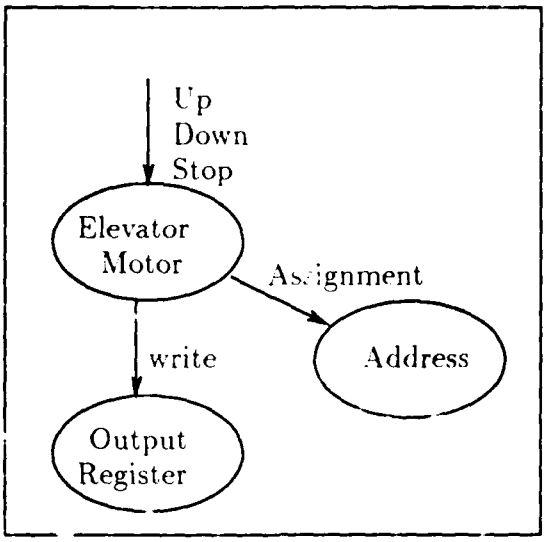


Figure A.42. Elevator Motor: Interface Diagram

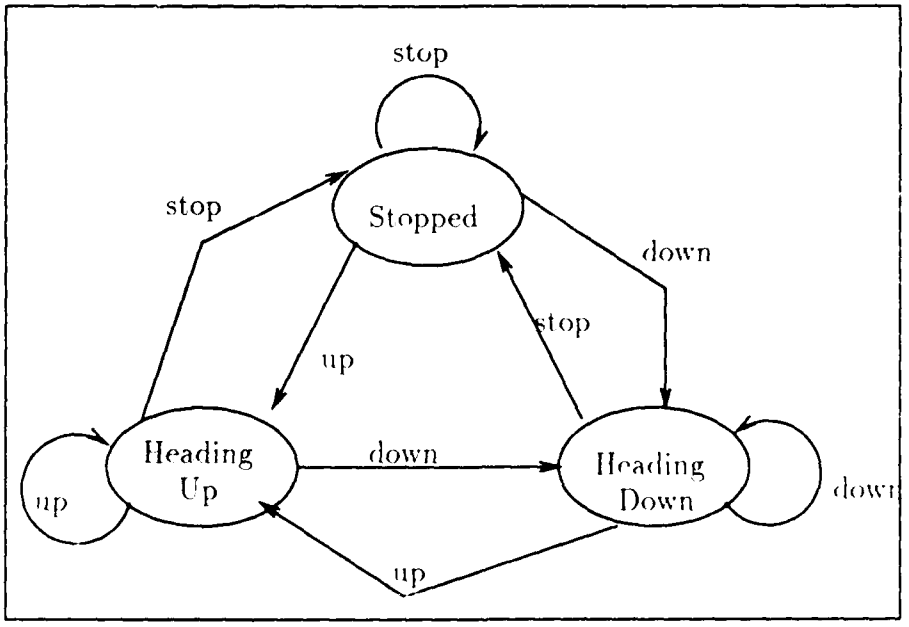


Figure A.43. Elevator Motor: State Transition Diagram

Floor

Textual Description:

This class of objects defines the floor numbers that an elevator can stop at. The floor number is implemented in eight bits so that it can fit in the input and output registers.

Structure Diagram, Interface Diagram, and State Transition Diagram:

See figures A.44, A.45, and A.46.

Messages received by class:

Assignment	Assign the value of one *floor* object to another.
Is Equal	Test to see if two *floor* objects have the same value.
Is Less Than	Test to see if one floor value is less than the other.

Messages sent by class:

None.

Description of any state limitations:

The value of an object of this class must be in the range 1..40. An object of this class may not be read until it has been assigned a value.

List of exported exceptions:

Constraint Error	The value assigned to an object of this class is not in the range 1..40.

List of exported constants:

Top Floor = 40

Re-use considerations:

This class is application specific.

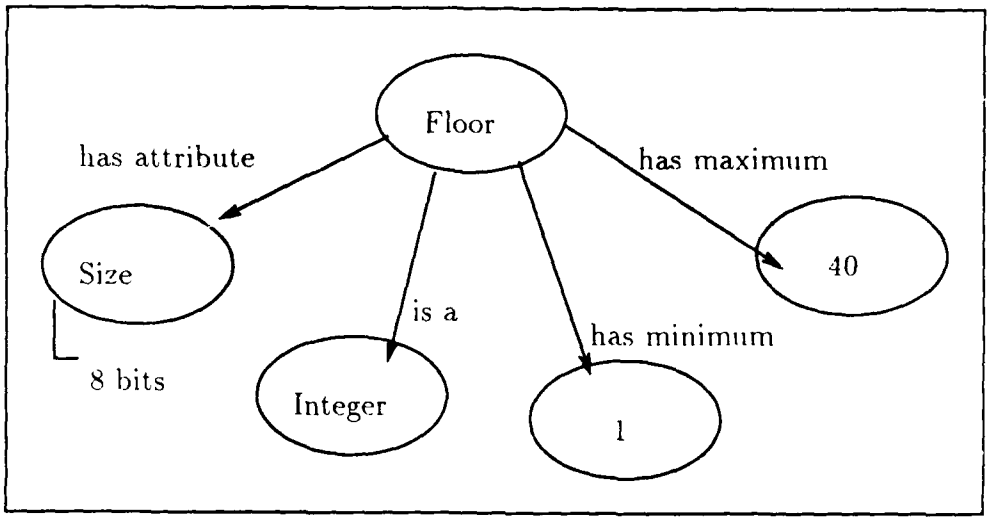


Figure A.44. Floor Number: Structure Diagram

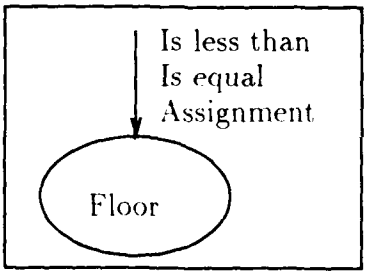


Figure A.45. Floor Number: Interface Diagram

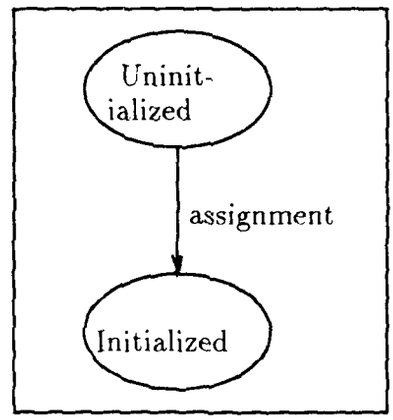


Figure A.46. Floor Number: State Transition Diagram

Floor Sensor

Textual Description:

This class of objects manages the elevator floor sensors. When the physical floor sensor triggers an interrupt and writes the floor number in an input register, this floor sensor manager reads the register and sends a message to some receiver. In this system, the receiver is always an elevator.

Structure Diagram, Interface Diagram, and State Transition Diagram:

See figures A.47, A.48, and A.49.

Messages received by class:

Arrival At Floor	This is a signal (interrupt) from the hardware that the elevator is approaching the floor whose number is in the input register.

Messages sent by class:

Address.Assignment	Assign the value of the input register address into it's object at initialization.
Input Register.Read	Read the floor number from the floor sensor input register.
Arrival At Floor	A message to the receiver that an elevator has reached a floor.
Floor.Assignment	Assign the value of one *floor* object to another.
Interrupt Number.Assignment	Assign the value of the floor sensor interrupt into its attribute at initialization.

Description of any state limitations:

None.

List of exported exceptions:

None.

List of exported constants:

None.

bf List of objects in class:

- Elevator 1 Floor Sensor
- Elevator 2 Floor Sensor
- Elevator 3 Floor Sensor
- Elevator 4 Floor Sensor

Re-use considerations:

This class has limited re-use potential.

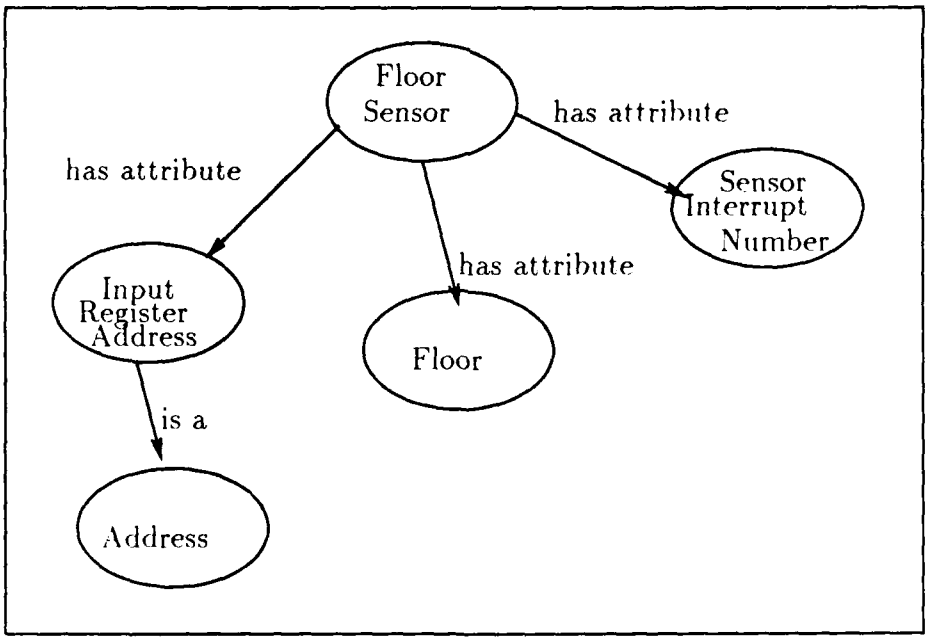


Figure A.47. Floor Sensor: Structure Diagram

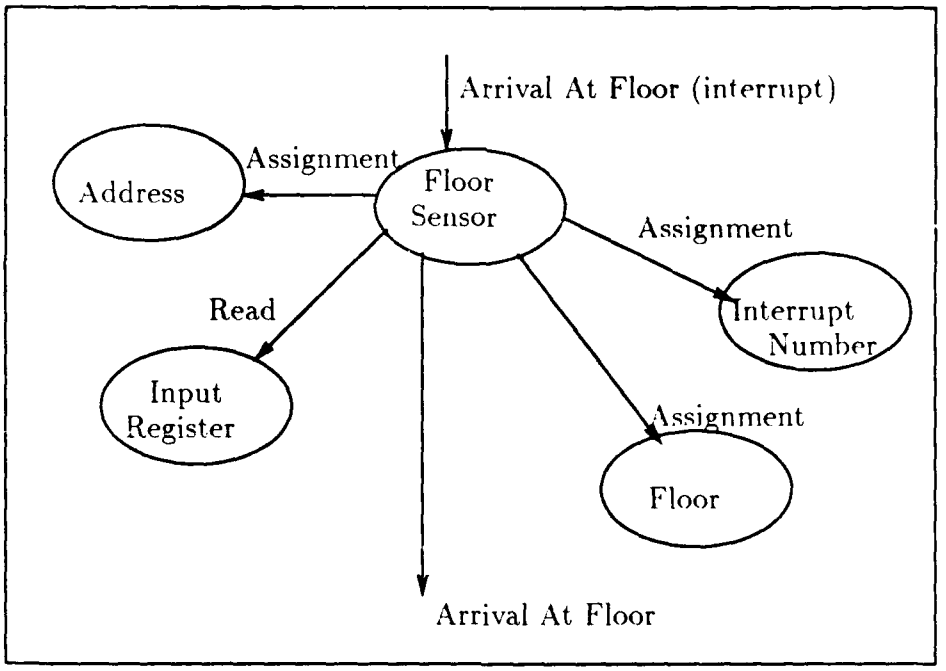


Figure A.48. Floor Sensor: Interface Diagram

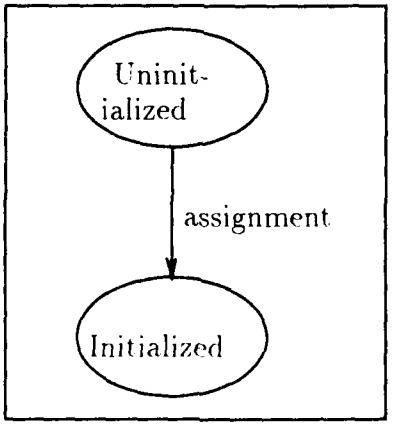


Figure A.49. Floor Sensor: State Transition Diagram

Input Register

Textual Description:

An input register is a hardware entity, but in many ways acts like a software entity. In this system, the input register will contain a bit pattern (eight bits) which can be interpreted as values of class *floor* or *weight* as appropriate when read by other objects.

Structure Diagram, Interface Diagram, and State Transition Diagram:

See figures A.50, and A.51. A state transition diagram is not appropriate for this class.

Messages received by class:

Read Read the value in the input register.

Messages sent by class:

None.

Description of any state limitations:

None.

List of exported exceptions:

N/A

List of exported constants:

N/A

List of objects in class:

- Elevator 1 Weight Sensor Register
- Elevator 2 Weight Sensor Register
- Elevator 3 Weight Sensor Register
- Elevator 4 Weight Sensor Register
- Elevator 1 Control Panel Input Register
- Elevator 2 Control Panel Input Register
- Elevator 3 Control Panel Input Register
- Elevator 4 Control Panel Input Register
- Elevator 1 Floor Sensor Input Register
- Elevator 2 Floor Sensor Input Register
- Elevator 3 Floor Sensor Input Register

- Elevator 4 Floor Sensor Input Register
- UP Summons Panel Input Register
- DOWN Summons Panel Input Register

Re-use considerations:

This is a description of a hardware entity and thus has no software re-use potential.

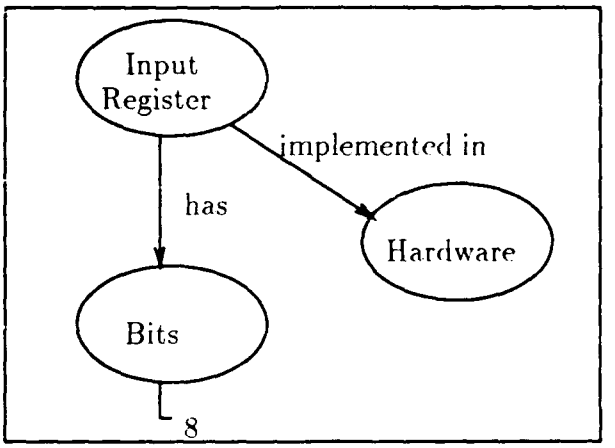


Figure A.50. Input Register: Structure Diagram

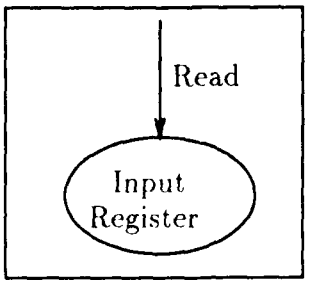


Figure A.51. Input Register: Interface Diagram

Interrupt Number

Textual Description:

This class defines the interrupt numbers. Valid interrupt numbers are those which can be represented in eight bits.

Structure Diagram, Interface Diagram, and State Transition Diagram:

See figures A.52, A.53, and A.54.

Messages received by class:

Assignment	Assign a value to an *Interrupt Number* object.

Messages sent by class:

None.

Description of any state limitations:

The value of an object of this class must be in the range 0..255. An object of this class may not be read until it has been assigned a value.

List of exported exceptions:

Constraint Error	The value assigned to an object of this class is not in the range 0..256.

List of exported constants:

None.

List of objects in class:

- Elevator 1 Control Panel Interrupt
- Elevator 2 Control Panel Interrupt
- Elevator 3 Control Panel Interrupt
- Elevator 4 Control Panel Interrupt
- Elevator 1 Floor Sensor Interrupt
- Elevator 2 Floor Sensor Interrupt
- Elevator 3 Floor Sensor Interrupt

- Elevator 4 Floor Sensor Interrupt
- UP Summons Request Panel Interrupt
- DOWN Summons Request Panel Interrupt

Re-use considerations:

This class has limited re-use potential.

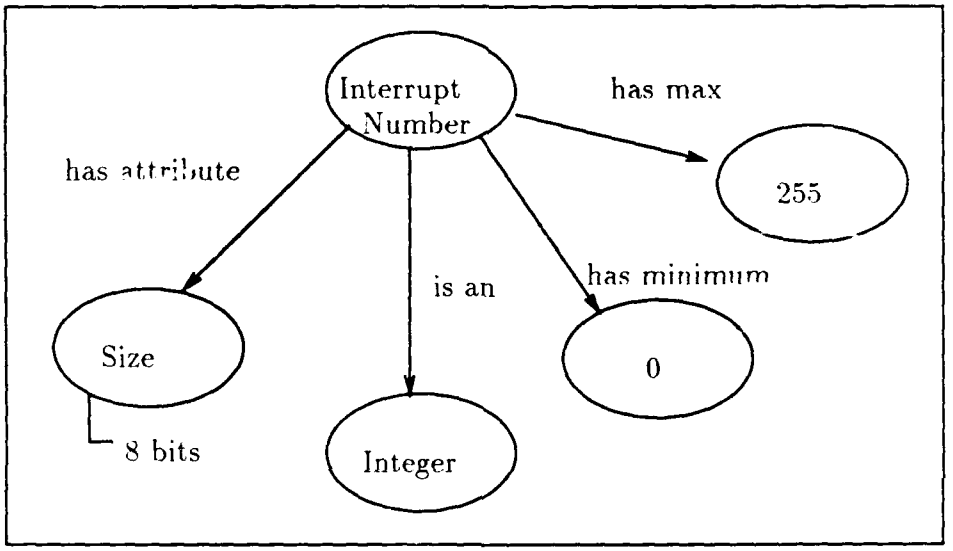


Figure A.52. Interrupt Number: Structure Diagram

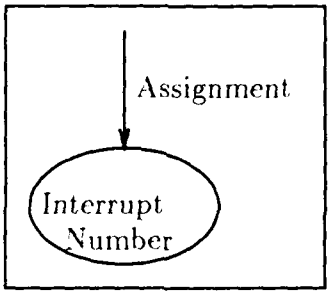


Figure A.53. Interrupt Number: Interface Diagram

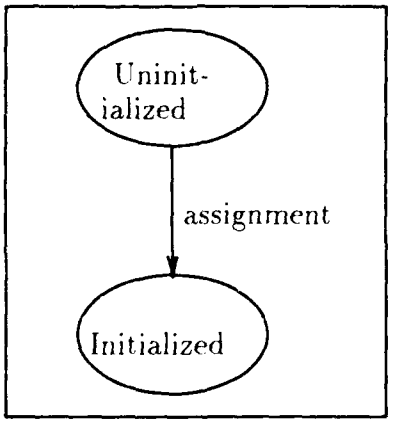


Figure A.54. Interrupt Number: State Transition Diagram

List

Textual Description:

The list class models a bounded list of items. Items can be added, or removed from the list, as well as tested for inclusion in the list.

Structure Diagram, Interface Diagram, and State Transition Diagram:

See figures A.55, A.56, and A.57.

Messages received by class:

Is Empty	Test to see if the list is empty of items.
Is In List	Test to see if a given item is in the list.
Add Item	Add an item to the list.
Remove Item	Remove an item from the list.

Messages sent by class:

Is Equal	Test if two items are equal (required from class of item).
Assignment	Assign the value of one item to another (required from class of item).

Description of any state limitations:

No item can be removed from an empty list, nor added to a full list.

List of exported exceptions:

Overflow	Attempt was made to add to a full list.
Underflow	Attempt was made to remove from an empty list.

List of exported constants:

Size — The size of the list.

Re-use considerations:

This class is potentially re-usable.

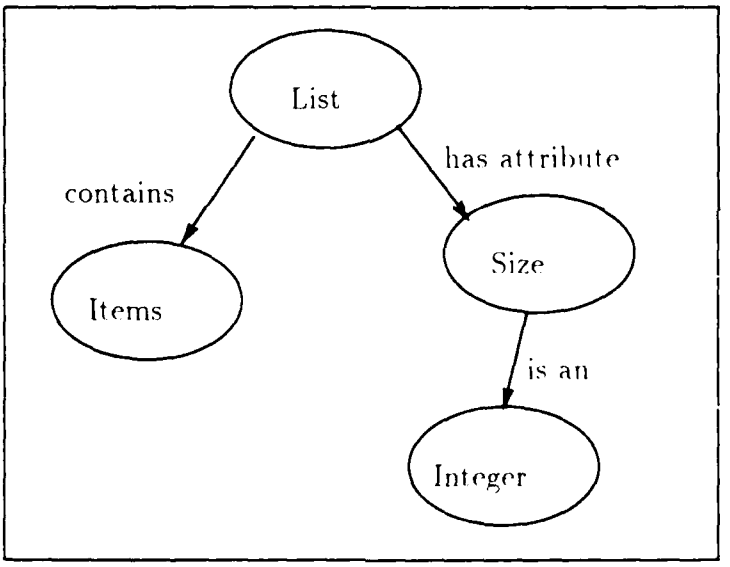


Figure A.55. List: Structure Diagram

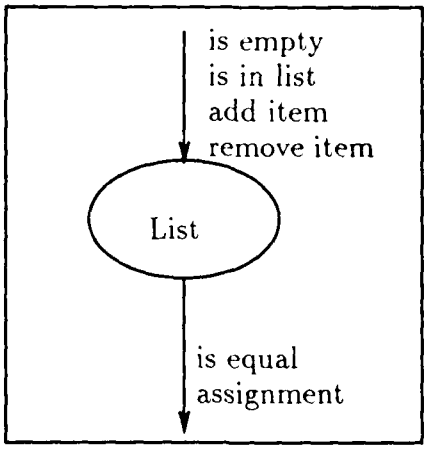


Figure A.56. List: Interface Diagram

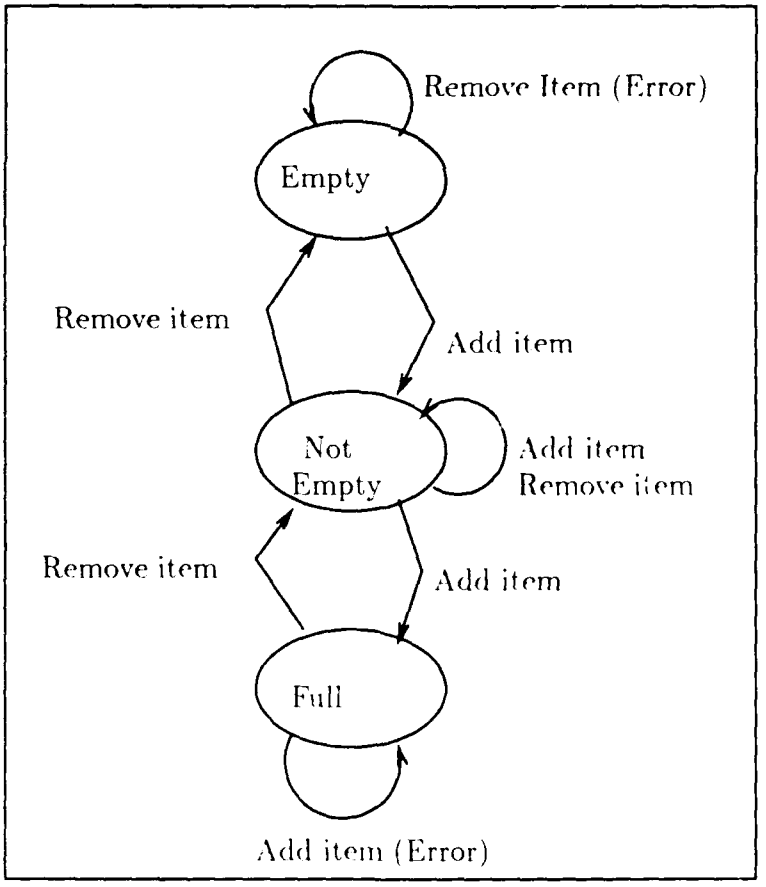


Figure A.57. List: State Transition Diagram

Location Panel

Textual Description:

This class controls the operation of the location panel hardware of an elevator. The class responds to messages to illuminate and extinguish lights in the panel. It does so by outputing the floor number of the light to the location panel output register. The class maintains a list of previously illuminated lamps (of which there *should* be only one). The location panel will only extinguish lights which were previously illuminated.

Structure Diagram, Interface Diagram, and State Transition Diagram:

See figures A.58, A.59, and A.60. The state transition diagram describes the activity of a single light on the panel.

Messages received by class:

Illuminate Light	Illuminate a new light on the location panel.
Extinguish Light	Extinguish a light on the location panel.

Messages sent by class:

Floor.Assignment	Assign a floor number value to the floor number attribute.
Address.Assignment	Assign the output register address to the attribute at initialization.
Output Register.Write	Write the floor number to the location panel output register to toggle the light.

List.Remove Item	Remove a floor number from the list of illuminated lights.
List.Add Item	Add a floor number to the list of illuminated lights.
List.Is In List	Test to see if a floor number is in the list of illuminated lights.

Description of any state limitations:

The location panel controller will check its status list before sending writing to the output port to toggle the light. Therefore, it will preclude the possibility of toggling the light off when it should be illuminating it, and vice versa.

List of exported exceptions:

None.

List of exported constants:

None.

List of objects in class:

- Elevator 1 Location Panel
- Elevator 2 Location Panel
- Elevator 3 Location Panel
- Elevator 4 Location Panel

Re-use considerations:

This class has limited re-use potential.

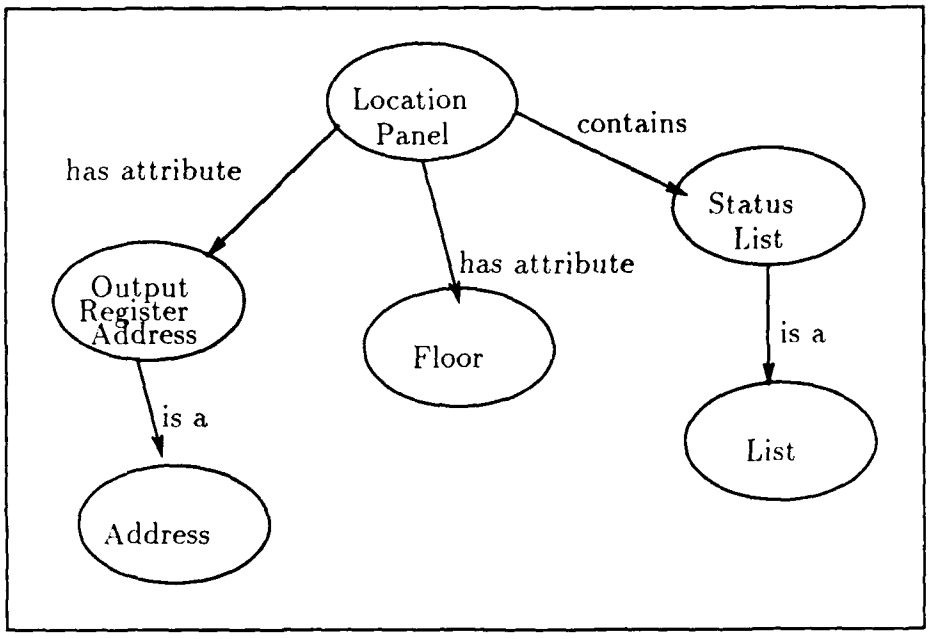


Figure A.58. Location Panel: Structure Diagram

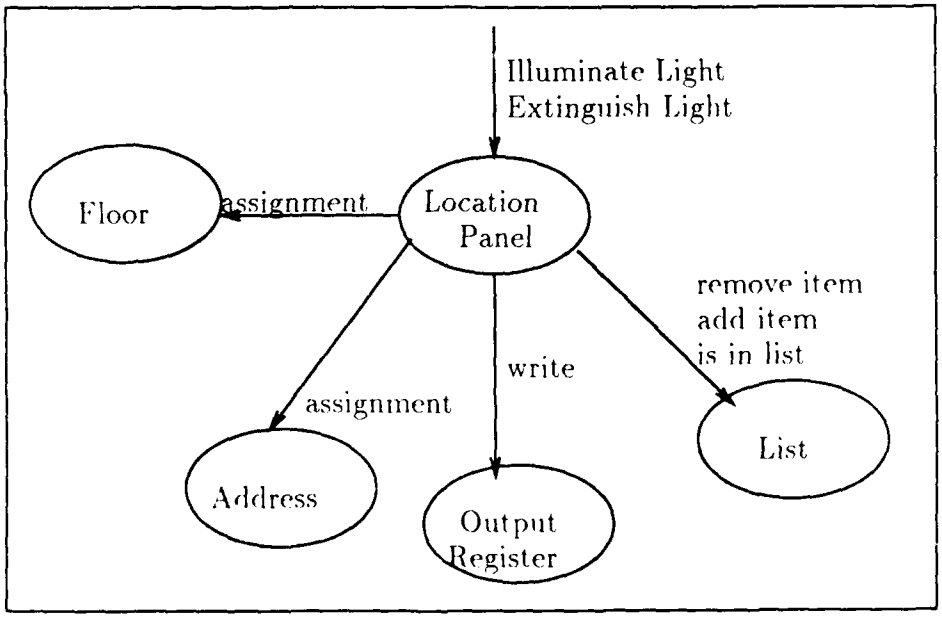


Figure A.59. Location Panel: Interface Diagram

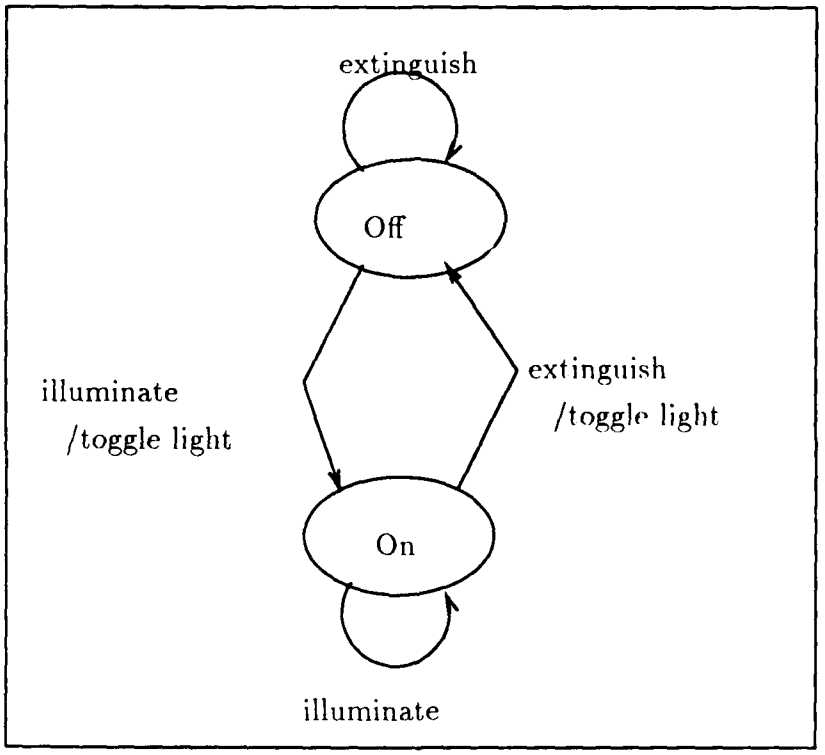


Figure A.60. Location Panel: State Transition Diagram

Output Register

Textual Description:

An output register is a hardware entity, but in many ways can be thought of as a software entity. In this system, the output register will be written a bit pattern (eight bits) which is interpreted by other hardware devices to be a floor number.

Structure Diagram, Interface Diagram, and State Transition Diagram:

See figures A.61 and ref5orid. A state transition diagram is not appropriate for this class.

Messages received by class:

Write Write the value into the output register.

Messages sent by class:

None.

Description of any state limitations:

None.

List of exported exceptions:

N/A

List of exported constants:

N/A

List of objects in class:

- Elevator 1 Location Panel Output Register
- Elevator 2 Location Panel Output Register
- Elevator 3 Location Panel Output Register
- Elevator 4 Location Panel Output Register
- Elevator 1 Control Panel Output Register
- Elevator 2 Control Panel Output Register
- Elevator 3 Control Panel Output Register

- Elevator 4 Control Panel Output Register
- Elevator 1 Motor Control Register
- Elevator 2 Motor Control Register
- Elevator 3 Motor Control Register
- Elevator 4 Motor Control Register
- Up Summons Panel Output Register
- DOWN Summons Panel Output Register

Re-use considerations:

This is a description of a hardware entity and thus has no software re-use potential.

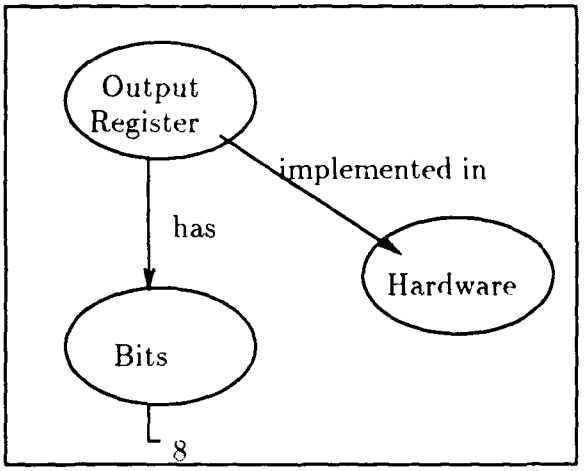


Figure A.61. Output Register: Structure Diagram

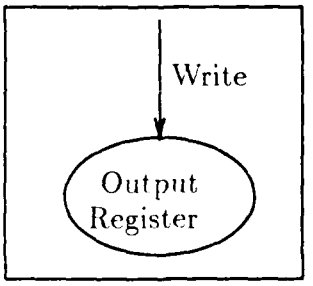


Figure A.62. Output Register: Interface Diagram

Summons Request

Textual Description:

The summons request class describes a structure which contains the important information about a summons request: its floor and direction. An object of this type is useful for placing in a list of requests for further processing.

Structure Diagram, Interface Diagram, and State Transition Diagram:

See figures A.63, A.64, and A.65.

Messages received by class:

Create	Create a summons request value from a floor number and direction.
Is Equal	Test the values of two summons request objects to see if they are equal.
Assignment	Assign a summons request value to another object.

Messages sent by class:

Floor.Assignment	Assign the floor number to the summons request component when creating a summons request.
Floor.Is Equal	Test to see if the floor number components of two summons requests are equal.
Direction.Assignment	Assign the direction to the summons request component when creating a summons request.
Direction.Is Equal	Test to see if the direction components of two summons requests are equal.

Description of any state limitations:

The floor number and direction components must each have legal values for those respective classes. An object of this class may not be read until it has been created via the *create* message.

List of exported exceptions:

Constraint Error	The value of the floor number or direction assigned to an object of this class is not in the proper range for its class.

List of exported constants:

None.

Re-use considerations:

This class is application specific.

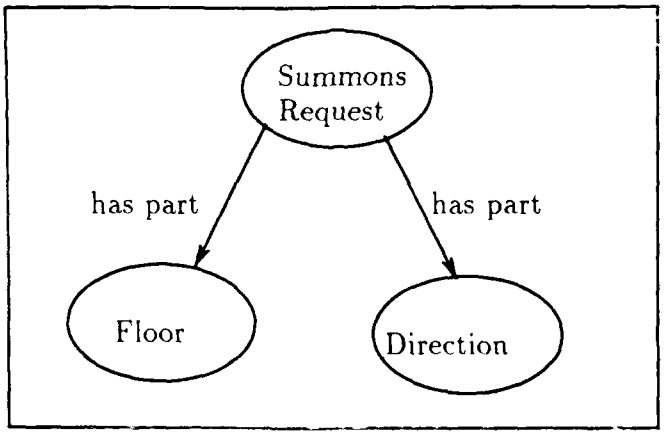


Figure A.63. Summons Request: Structure Diagram

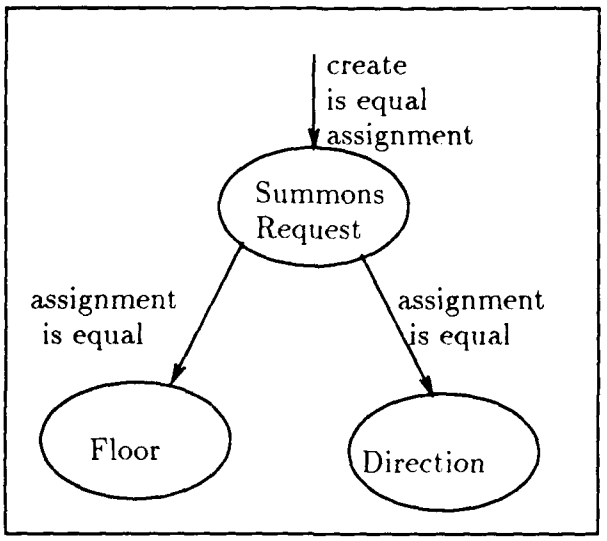


Figure A.64. Summons Request: Interface Diagram

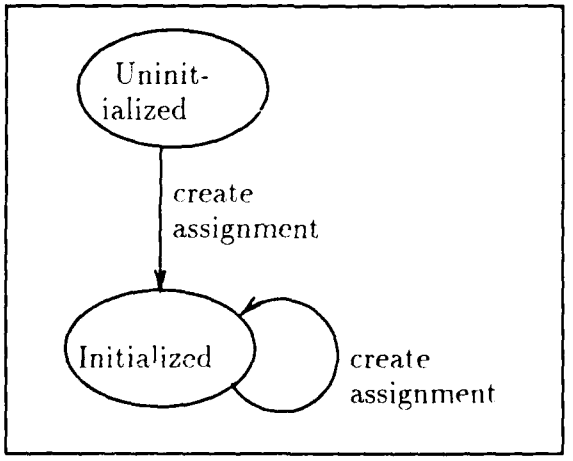


Figure A.65. Summons Request: State Transition Diagram

Weight

Textual Description:

This class of objects defines the weight of an entity, in this case elevators. The units for this class are in hundreds of pounds. The type must be implemented in eight bits, since its value is taken from an input register.

Structure Diagram, Interface Diagram, and State Transition Diagram:

See figures A.66, and A.67. A state transition diagram is not required for this class.

Messages received by class:

Assignment	Assign a *weight* value to an object.
Is Less Than	Test to see if one value of *weight* is less than another.

Messages sent by class:

None.

Description of any state limitations:

The value of an object of this class must be in the range 0..255. An object of this class may not be read until it has been assigned a value.

List of exported exceptions:

Constraint Error	The value assigned to an object of this class is not in the range 0..255.

List of exported constants:

Max Weight — 255

Re-use considerations:

This class is application specific.

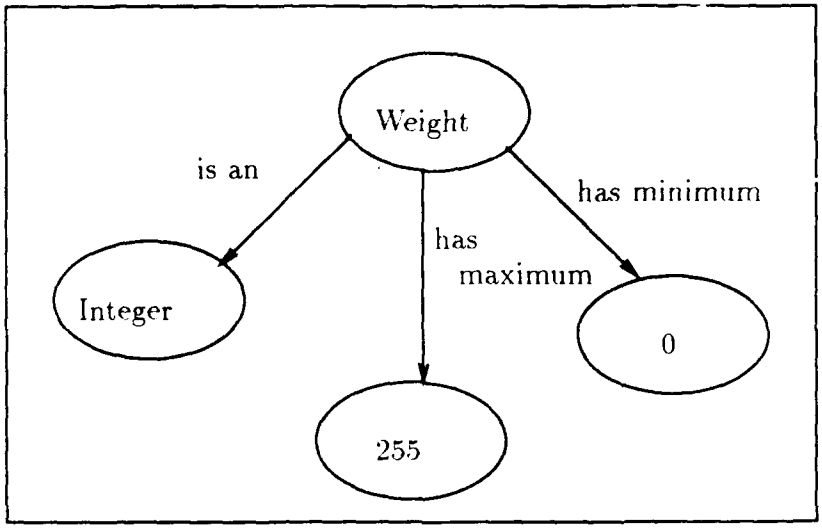


Figure A.66. Weight: Structure Diagram

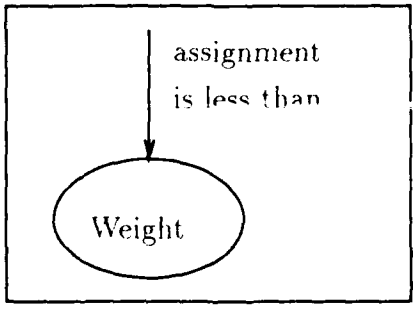


Figure A.67. Weight: Interface Diagram

Weight Sensor

Textual Description:

This class of objects manages the weight sensor for an elevator. The physical weight sensor periodically places a weight value in an input register. When a weight check is requested, a weight sensor object will read the weight sensor input register and return the weight value.

Structure Diagram, Interface Diagram, and State Transition Diagram:

See figures A.68, A.69, and A.70.

Messages received by class:

Check Weight	Read the weight sensor input register and return the weight value.

Messages sent by class:

Weight.Assignment	Assign the weight value from the input register to the current weight attribute.
Input Register.Read	Read the weight value from the weight sensor input register.
Address.Assignment	Assign the address of the input register to its attribute at initialization.

Description of any state limitations:

None.

List of exported exceptions:

None.

List of exported constants:

None.

List of objects in class:

- Elevator 1 Weight Sensor
- Elevator 2 Weight Sensor
- Elevator 3 Weight Sensor
- Elevator 4 Weight Sensor

Re-use considerations:

This class has limited re-use potential.

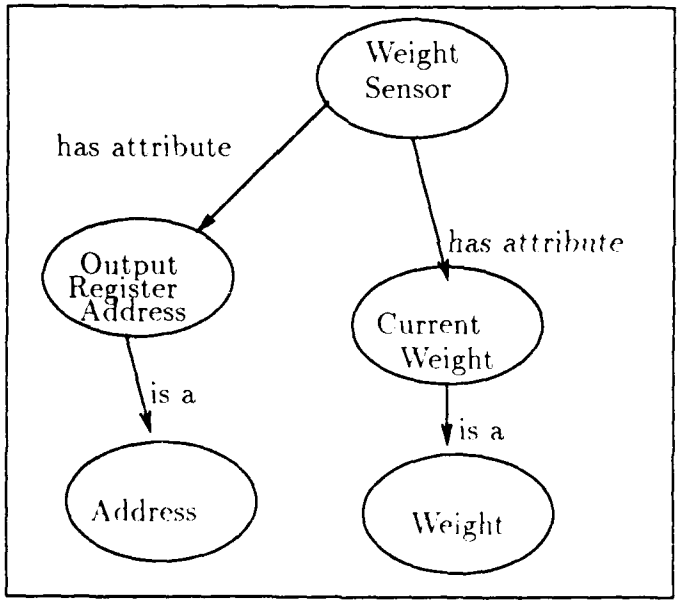


Figure A.68. Weight Sensor: Structure Diagram

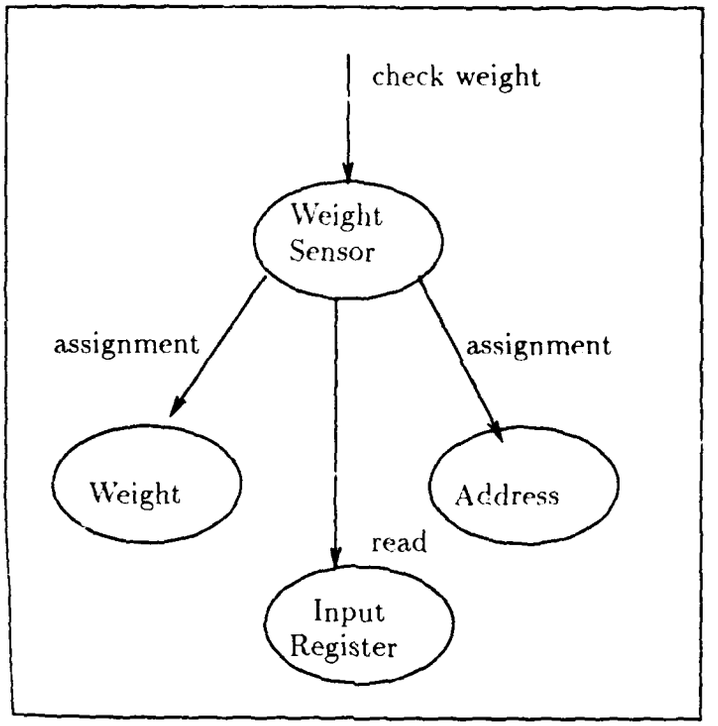


Figure A.69. Weight Sensor: Interface Diagram

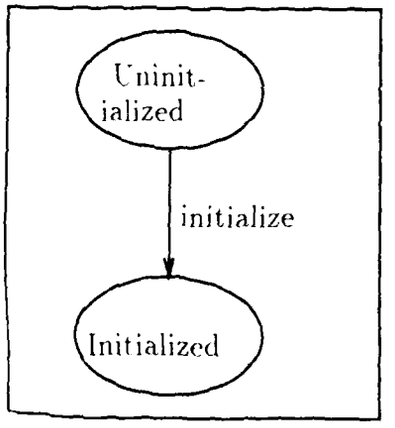


Figure A.70. Weight Sensor: State Transition Diagram

Bibliography

Abbott, Russel J. ""Program Design by Informal English Descriptions"," *Communications of the ACM*, *26*(11):882–894 (November 1983).

Barnes, Patrick D. *A Decision-Based Methodology for Object Oriented Design*. MS thesis, AFIT/GCS/ENG/88D-1, Air Force Institute of Technology, 1988 (ADA202579).

Booch, Grady. *Software Engineering with Ada*. The Benjamin/Cummings Publishing Company, Inc., 1983.

Booch, Grady. "Object Oriented Development," *IEEE Transactions on Software Engineering*, *SE-12*(2):211–221 (February 1986).

Booch, Grady. *Software Components with Ada*. The Benjamin/Cummings Publishing Company, Inc., 1987.

Booch, Grady. *Software Engineering with Ada* (2nd Edition). The Benjamin/Cummings Publishing Company, Inc., 1987.

Bralick Jr., William A. *An Examination of the Theoretical Foundations of the Object-Oriented Paradigm*. MS thesis, AFIT/GCS/MA/88M-01, Air Force Institute of Technology, 1988 (ADA194879).

Chen, Peter P. ""The Entity-Relationship Model–Toward a Unified View of Data"," *ACM Transactions on Database Systems*, *1*(1):9–36 (March 1976).

Coad Jr., Peter. "Object Oriented Requirements Analysis (OORA)." In *Proceedings of the Twelfth Annual International Computer Software and Applications Conference*, page 436, 1988.

DeMarco, Tom. *Structured Analysis and System Specification*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1979.

Department of Defense. *Reference Manual for the Ada Programming Language*. *ANSI/MIL-STD-1815A*, 1983.

EVB Software Engineering Inc. *An Object Oriented Design Handbook for Ada Software*. Frederick, MD, 1985.

EVB Software Engineering Inc. *Object Oriented Requirements Analysis*. Frederick, MD, 1989.

Gane, Chris and Trish Sarson. *Structured Systems Analysis: Tools and Techniques*. St. Louis: McDonnell Douglas, 1982.

Gomaa, Hassan and Douglas B. H. Scott. "Prototyping as a Tool in the Specification of User Requirements." In *Proceedings of the IEEE Fifth Annual International Conference on System Engineering*, pages 333–342, 1981.

Ichbiah, Jean D., et al. *Rationale for the Design of the Ada Programming Language*, 1986.

ANSI/IEEE Standard 729-1983. IEEE Standard Glossary of Software Engineering Terminology.

Jorgensen, Paul C. "Tutorial On Requirements Specification." In *Proceedings of the IEEE Computer Society's Tenth Annual International Computer Software and Applications Conference*, page 182, 1986.

Kenth, Norman L., et al. "Summary of Discussions from OOPSLA-87's Methodologies and OOP Workshop." In *Addendum to the Proceedings OOPSLA '87. ACM SIGPLAN Notices*, 1987.

Ladden, Richard M. "A Survey of Issues to be Considered in the Development of an Object-Oriented Development Methodology for Ada," *ACM Ada Letters*, *9*(2):78–89 (March/April 1989).

Land, F.F. and M. Kennedy-McGregor. "Information and Information Systems: Concepts and Perspectives." In Galliers, Robert, editor, *Information Analysis: Selected Readings*, pages 63–91, Addison-Wesley Publishers Ltd., 1987.

Land, Frank. "Adapting to Changing User Requirements." In Galliers, Robert, editor, *Information Analysis: Selected Readings*, pages 203–229, Addison-Wesley Publishers Ltd., 1987.

MacLennan, Bruce J. *Principles of Programming Languages: Design, Evaluation, and Implementation*. New York: CBS College Publishing, 1983.

Martin, James and Carma McClure. *Structured Techniques for Computing*. Prentice-Hall, Inc., 1985.

McFarren, Michael R. *Using Concept Mapping to Define Proplems and Identify Key Kernels During the Development of a Decision Support System*. MS thesis, AFIT/GST/ENS/87M-12, Air Force Institute of Technology, 1987 (ADA185636).

McMenamin, Stephen M. and John F. Palmer. *Essential Systems Analysis*. New York: Yourdon Inc., 1984.

Mittermeir, Roland T., et al. "Alternatives to Overcome the Communication Problems of Formal Requirements Analysis." In Galliers, Robert, editor, *Information Analysis: Selected Readings*, pages 153–165, Addison-Wesley Publishers Ltd., 1987.

Novak, Joseph D. and D. Bob Gowin. *Learning How to Learn*. Cambridge University Press, 1984.

Page-Jones, Meilir. *Practical Project Management: Restoring Quality to DP Projects and Systems*. New York: Dorset House Publishing, 1985.

Parnas, David L. "On the Criteria To Be Used in Decomposing Systems into Modules," *Communications of the ACM*, *15*(12):1053-1058 (December 1972).

Pascoe, Geoffrey A. "Elements of Object-Oriented Programming," *Byte*, *11*(8):139-144 (August 1986).

Peters, Lawrence. *Advanced Structured Analysis and Design*. Englewood Cliffs, NJ: Prentice-Hall, 1987.

Pressman, Roger S. *Software Engineering: A Practitioner's Approach* (2nd Edition). McGraw-Hill, Inc., 1987.

Ross, Douglas T. and Kenneth E. Schoman, Jr. "Structured Analysis for Requirements Definition," *IEEE Transactions on Software Engineering*, *SE-3*(1):6-15 (January 1977).

Seidewitz, Ed and Mike Stark. *General Object-Oriented Software Development*. Technical Report SEL-86-002, NASA Goddard Space Flignt Center, VA, 1986.

Seidewitz, Ed and Mike Stark. "Towards a General Object-Oriented Software Development Methodology," *ACM SIGAda Ada Letters*, *7*(4):4.54-4.67 (July-August 1987).

Shlaer, Sally and Stephen J. Mellor. "Three Approaches to System Analysis," *Computer Design*, *27*(1):55 (January 1988).

Sprague, Ralph and Eric D. Carlson. *Building Effective Decision Support Systems*. Englewood Cliffs, NJ: Prentice-Hall Inc., 1982.

Umphress, David A., "Object Oriented Requirements Analysis," 1988. Class notes for MATH 555, Introduction to Software Engineering with Ada, at the Air Force Institute of Technology (AFIT).

Valusek, John R. and Dennis G. Fryback. "Information Requirements Determination: Obstacles Within, Among and Between Participants." In Galliers, Robert, editor, *Information Analysis: Selected Readings*, pages 139-151, Addison-Wesley Publishers Ltd., 1987.

Ward, Paul T. "How to Integrate Object Orientation with Structured Analysis and Design," *IEEE Software*, pages 74-82 (March 1989).

Yadav, Surya B., et al. "Comparison of Analysis Techniques for Information Requirements Determination," *Communications of the ACM*, *31*(9):1090-1097 (September 1988).

Yourdon, Edward. *Modern Structured Analysis*. Prentice-Hall, Inc., 1989.

Vita

Capt Steven G. March [redacted] graduating as valedictorian in 1981. Capt March entered the United States Air Force Academy that year, where he majored in Computer Science. In May of 1985, Capt March was awarded a Bachelor of Science degree as a distinguished graduate, and commissioned as an Air Force officer.

Capt March was then assigned to the 3390 Technical Training Group (Keesler AFB, MS) where he served for three years as an Ada Software Engineering Instructor. Capt March entered the Air Force Institute of Technology, School of Engineering, in May of 1988.

[redacted]

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) AFIT/GCS/ENC/89D-1	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION School of Engineering 6b. OFFICE SYMBOL (If applicable) AFIT/ENA	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS (City, State, and ZIP Code) Air Force Institute of Technology Wright-Patterson AFB, Ohio 45433	7b. ADDRESS (City, State, and ZIP Code)
8a. NAME OF FUNDING/SPONSORING ORGANIZATION SDIO Phase 1 Program Office 8b OFFICE SYMBOL (If applicable) S/PI	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code) The Pentagon Washington, D.C. 20301-7100	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO / WORK UNIT ACCESSION NO

11. TITLE (Include Security Classification)
AN OBJECT ORIENTED ANALYSIS METHOD FOR Ada AND EMBEDDED SYSTEMS - UNCLASSIFIED

12. PERSONAL AUTHOR(S)
Steven G. March, Capt, USAF

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
MS Thesis	FROM _____ TO _____	1989 December	240

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	
12	05		Software Engineering, Computer systems analysis Computer program documentation

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Thesis Chairman: David A. Umphress, Maj, USAF
Assistant Professor of Mathematics and Computer Science

Abstract: See Reverse

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	UNCLASSIFIED
22a NAME OF RESPONSIBLE INDIVIDUAL Maj David A. Umphress	22b TELEPHONE (Include Area Code) (513) 255-3098 / 22c OFFICE SYMBOL AFIT/ENC

DD Form 1473, JUN 86 *Previous editions are obsolete.*

Object-Oriented Design (OOD) has become a popular approach to software development with Ada. One of the difficulties in applying OOD is that the information available to the designer (the product of requirements analysis) is typically presented in a form inappropriate to OOD. Traditional requirements analysis tools (e.g. data flow diagrams) organize the software requirements based upon the functions the system must perform. Recent research suggests that an object-oriented approach to requirements analysis is a more natural lead-in to OOD.

The goal of this thesis was to define the tools, steps, and heuristics for an object-oriented analysis (OOA) method of modeling software requirements. The choice of tools used to capture the requirements makes the method particularly suitable for use when developing embedded systems. The method emphasizes communication with both the domain expert and the designer.

The OOA method consists of two phases. The objective of the firse phase is to capture the software requirements using unstructured tools such as concept maps, storyboards, and a list of external events to which the system must respond. The second phase involves structuring these requirements into a model based upon the software objects.

The thesis also addressed the possibility of automated support for the OOA method, and proposes an OOA tool to assist the analyst. The OOA method was applied to a sample requirements analysis problem to demonstrate the method's feasibility.